INFORMIX®-4GL

Reference

Version 6.0
April 1994
Part No. 000-7611

Published by INFORMIX® Press Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

The following are worldwide trademarks of Informix Software, Inc., or its subsidiaries, registered in the United States of America as indicated by an "®," and in numerous other countries worldwide:

INFORMIX® and **C-ISAM**®.

The following are worldwide trademarks of the indicated owners or their subsidiaries, registered in the United States of America as indicated by an "®," and in numerous other countries worldwide:

X/OpenCompany Ltd.: UNIX®; X/Open®
Adobe Systems Incorporated: Post Script®

Some of the products or services mentioned in this document are provided by companies other than Informix. These products or services are identified by the trademark or servicemark of the appropriate company. If you have a question about one of those products or services, please call the company in question directly.

ACKNOWLEDGMENTS

The following people contributed to this version of *INFORMIX-4GL Reference*:

Documentation Team: Adam Barnett, Diana Boyd, Kaye Bonney, Lisa Braz, Mitch Gordon, Tom Houston, Todd Katz, Liz Knittel, Dawn Maneval, Sara Odom

Technical Contributors: Alan Denney, Jonathan Leffler, Kevin Rowney

Copyright © 1981-1994 by Informix Software, Inc. All rights reserved.

No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems—without permission of the publisher.

RESTRICTED RIGHTS LEGEND

Software and accompanying materials acquired with United States Federal Government funds or intended for use within or for any United States federal agency are provided with "Restricted Rights" as defined in DFARS 252.227-7013(c)(1)(ii) or FAR 52.227-19.

Preface

This *INFORMIX-4GL Reference* manual is a complete guide to the features and syntax of the **INFORMIX-4GL** language.

You do not need database management experience, nor familiarity with relational database concepts, to use this manual. A knowledge of SQL (Structured Query Language), however, and experience using a high-level programming language would be useful. Concepts underlying the **INFORMIX-4GL** language are described in a companion volume, *INFORMIX-4GL Concepts and Use*.

Informix database engines and the SQL language are described in separate manuals, including *Informix Guide to SQL: Tutorial*, and *Informix Guide to SQL: Reference*.

Summary of Chapters

This *INFORMIX-4GL Reference* manual includes the following chapters and appendices:

- This Preface provides general information about the manual and lists additional references to help you understand **INFORMIX-4GL** concepts.
- The Introduction describes the documentation set of **INFORMIX-4GL**, explains how to read syntax diagrams, and describes some features of **INFORMIX-4GL**.
- Chapter 1, "Compiling INFORMIX-4GL Source Files," describes the *C Compiler* and *Rapid Development System* implementations of **INFORMIX-4GL**. It also explains how to create executable versions of **4GL** source files, both from the Programmer's Environment and the command line.
- Chapter 2, "The INFORMIX-4GL Language," provides an overview of **4GL** language features and visual features of the applications that you can create with **INFORMIX-4GL**.

- Chapter 3, "INFORMIX-4GL Statements," describes the statements of **4GL** in alphabetical order. Additional sections describe **4GL** data types, expressions, and other syntax topics that affect several statements.
- Chapter 4, "Built-In Functions and Operators," includes an overview of the predefined functions and operators of **4GL**, and describes the syntax of each built-in function and built-in operator, with examples of usage.
- Chapter 5, "Screen Forms," provides an overview of **4GL** screen forms and form drivers, and describes the syntax of **4GL** form specification files. It also describes how to create default forms with the Form Compiler tool, and how the Column Attributes Dictionary sets default attributes.
- Chapter 6, "INFORMIX-4GL Reports," offers an overview of **4GL** reports and report drivers, and describes the syntax of **4GL** report definitions. It also describes the syntax of statements and operators that can appear only in **4GL** reports.
- Appendix A, "The Demonstration Database and Application," describes the structure and content of the tables in the stores demonstration database.
- Appendix B, "INFORMIX-4GL Utility Programs," describes the **mkmessage** and **upscol** utility programs.
- Appendix C, "Using C with INFORMIX-4GL," describes how to call C functions from **4GL** programs, and vice versa, and describes a function library for conversion between the DECIMAL data type of 4GL and the C data types.
- Appendix D, "Environment Variables," describes the environment variables that are used by **INFORMIX-4GL**.
- Appendix E, "Native Language Support Within INFORMIX-4GL," describes how the NLS environment variables affect your **4GL** programs.
- Appendix F, "Modifying termcap and terminfo," describes the modifications you can make to your termcap and terminfo files to extend function key definitions, to specify characters for window borders, and to enable **INFORMIX-4GL** programs to interact with terminals that support color displays.
- Appendix G, "The ASCII Character Set," lists the ASCII characters and their numeric codes.
- Appendix H, "Reserved Words," lists reserved words of **INFORMIX-4GL**.
- Appendix I, "Developing Applications for International Markets," describes the internationalization features provided with **4GL** and shows how to develop **4GL** applications that are world-ready and easy to localize.

- The Glossary defines terms used throughout the **4GL** documentation set.
- The Index lists page references to selected topics in this manual.

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
[TETC Technical Publications Department]
4100 Bohannon Drive
Menlo Park, CA 94025

If you prefer to send electronic mail, our address is:

doc@informix.com

We appreciate your feedback.

Related Reading

If you have no prior experience with database management, you should refer to the *Informix Guide to SQL: Tutorial.* This manual is provided with all Informix database servers.

For additional technical information on database management, consult the following texts by C. J. Date:

- *Database: A Primer* (Addison-Wesley Publishing, 1983)
- *An Introduction to Database Systems, Volume I* (Addison-Wesley Publishing, 1990)
- *An Introduction to Database Systems, Volume II* (Addison-Wesley Publishing, 1983)

This guide assumes that you are familiar with the UNIX operating system. If you have limited UNIX experience, you might want to look at your operating system manual or a good introductory text before you read this manual.

Some suggested texts about UNIX systems follow:

- *A Practical Guide to the UNIX System*, Second Edition, by M. Sobell (Benjamin/Cummings Publishing, 1989)
- *A Practical Guide to UNIX System V* by M. Sobell (Benjamin/Cummings Publishing, 1985)
- *Introducing the UNIX System* by H. McGilton and R. Morgan (McGraw-Hill Book Company, 1983)
- *UNIX for People* by P. Birns, P. Brown, and J. Muster (Prentice-Hall, 1985)

If you are interested in learning more about the SQL language, consider the following text:

- *Using SQL* by J. Groff and P. Weinberg (Osborne McGraw-Hill, 1990)

INFORMIX-4GL Reference

Preface

Summary oaf Chapters iii
Informix Welcomes Your Comments v
Related Reading v

Introduction

Documentation Included with 4GL 4
Other Useful Documentation 5
Conventions of this Manual 5
  Typographical Conventions 5
  Syntax Notation 6
Useful On-Line Files 10
On-Line Error Messages 11
The stores Demonstration Application and Database 12
New Features in 4GL 12
  NLS Support 12
  Improved Performance 12
  Improved Quality 13
Accessing Databases from Within 4GL 13
  Preparing SQL Statements 13
Compatibility and Migration 14

Chapter 1 **Compiling INFORMIX-4GL Source Files**

Chapter Overview 1-3
Two Implementations of INFORMIX-4GL 1-3
  Differences Between the C Compiler and RDS Versions 1-4
The C Compiler Version 1-6
  The Programmer´s Environment 1-6
  Creating Programs in the Programmer's Environment 1-23
  Creating Programs at the Command Line 1-27
  Program Filename Extensions 1-32

The Rapid Development System Version 1-34
The Programmer´s Environment 1-34
Creating Programs in the Programmer’s Environment 1-52
Creating Programs at the Command Line 1-56
Program Filename Extensions 1-71

Chapter 2 **The INFORMIX-4GL Language**
Overview of 4GL 2-3
Language Features 2-3
Lettercase Insensitivity 2-3
4GL Statements 2-4
Comments 2-5
Source Code Modules and Program Blocks 2-7
Statement Blocks 2-8
Statement Segments 2-9
4GL Identifiers 2-9
Interacting with Users 2-15
Ring Menus 2-15
Screen Forms 2-17
4GL Windows 2-19
On-Line Help 2-21
Exception Handling 2-23
Error Handling with SQLCA 2-23
A Taxonomy of Run-Time Errors 2-26

Chapter 3 **INFORMIX-4GL Statements**
Chapter Overview 3-11
The 4GL Statement Set 3-11
Types of SQL Statements 3-11
Other Types of 4GL Statements 3-13
Statement Descriptions 3-15
CALL 3-16
CASE 3-21
CLEAR 3-26
CLOSE FORM 3-29
CLOSE WINDOW 3-30
CONSTRUCT 3-31
CONTINUE 3-55
CURRENT WINDOW 3-56
DATABASE 3-58
DEFER 3-62
DEFINE 3-65
DISPLAY 3-74

DISPLAY ARRAY 3-85
DISPLAY FORM 3-93
END 3-95
ERROR 3-96
EXIT 3-98
FINISH REPORT 3-100
FOR 3-102
FOREACH 3-105
FUNCTION 3-111
GLOBALS 3-117
GOTO 3-122
IF 3-124
INITIALIZE 3-125
INPUT 3-128
INPUT ARRAY 3-152
LABEL 3-177
LET 3-178
LOAD 3-181
LOCATE 3-186
MAIN 3-191
MENU 3-193
MESSAGE 3-213
NEED 3-216
OPEN FORM 3-217
OPEN WINDOW 3-219
OPTIONS 3-228
OUTPUT TO REPORT 3-242
PAUSE 3-244
PREPARE 3-245
PRINT 3-254
PROMPT 3-255
REPORT 3-260
RETURN 3-263
RUN 3-265
SCROLL 3-268
SKIP 3-269
SLEEP 3-270
START REPORT 3-271
UNLOAD 3-274
VALIDATE 3-278
WHENEVER 3-281
WHILE 3-287

Statement Segments 3-289
- ATTRIBUTE 3-290
- Color and Monochrome Attributes 3-291
- Precedence of Attributes 3-292
- Data Types of 4GL 3-293
- The Simple Data Types 3-294
- The Structured Data Types 3-296
- The Large Data Types 3-296
- Descriptions of the 4GL Data Types 3-296
- ARRAY 3-297
- BYTE 3-298
- CHAR 3-299
- CHARACTER 3-300
- DATE 3-300
- DATETIME 3-300
- DEC 3-304
- DECIMAL 3-304
- DOUBLE PRECISION 3-305
- FLOAT 3-305
- INT 3-306
- INTEGER 3-306
- INTERVAL 3-307
- MONEY 3-312
- NUMERIC 3-313
- REAL 3-313
- RECORD 3-313
- SMALLFLOAT 3-315
- SMALLINT 3-316
- TEXT 3-317
- VARCHAR 3-318
- Data Type Conversion 3-319
- Summary of Compatible 4GL Data Types 3-324
- Expressions of 4GL 3-326
- Components of 4GL Expressions 3-327
- 4GL Boolean Expressions 3-333
- Integer Expressions 3-338
- Number Expressions 3-341
- Character Expressions 3-343
- Time Expressions 3-347
- Field Clause 3-359
- Table Qualifiers 3-361
- THRU or THROUGH Keywords and .* Notation 3-363

Chapter 4 Built-In Functions and Operators

Functions in 4GL Programs 4-5
- Built-In 4GL Functions 4-5
- Built-In SQL Functions 4-6
- C Functions 4-6
- ESQL/C Functions 4-7
- Programmer-Defined 4GL Functions 4-7
- Invoking Functions 4-8

Operators of 4GL 4-10

Syntax of Built-In Functions and Operators 4-11
- Aggregate Report Functions 4-13
- ARG_VAL( ) 4-16
- Arithmetic Operators 4-18
- ARR_COUNT( ) 4-24
- ARR_CURR( ) 4-26
- ASCII 4-28
- Boolean Operators 4-30
- CLIPPED 4-38
- COLUMN 4-40
- CURRENT 4-42
- DATE 4-44
- DATE( ) 4-45
- DAY( ) 4-46
- DOWNSHIFT( ) 4-47
- ERR_GET( ) 4-48
- ERR_PRINT( ) 4-49
- ERR_QUIT( ) 4-50
- ERRORLOG( ) 4-51
- EXTEND( ) 4-53
- FGL_DRAWBOX( ) 4-56
- FGL_GETENV( ) 4-58
- FGL_KEYVAL( ) 4-60
- FGL_LASTKEY ( ) 4-62
- FIELD_TOUCHED( ) 4-64
- GET_FLDBUF( ) 4-66
- INFIELD( ) 4-69
- LENGTH( ) 4-71
- LINENO 4-73
- MDY( ) 4-74
- MONTH( ) 4-75
- NUM_ARGS( ) 4-76
- PAGENO 4-77
- SCR_LINE( ) 4-78

SET_COUNT( ) 4-80
SHOWHELP( ) 4-81
SPACE 4-82
SQLEXIT( ) 4-83
STARTLOG( ) 4-84
TIME 4-86
TODAY 4-87
UNITS 4-89
UPSHIFT( ) 4-90
USING 4-91
WEEKDAY( ) 4-100
WORDWRAP 4-102
YEAR( ) 4-104

Chapter 5 **Screen Forms**
4GL Forms 5-3
Form Drivers 5-3
Form Fields 5-4
Structure of a Form Specification File 5-6
DATABASE Section 5-10
Database References in the DATABASE Section 5-11
The FORMONLY Option 5-11
The WITHOUT NULL INPUT Option 5-12
SCREEN Section 5-12
The SIZE Option 5-13
The Screen Layout 5-14
Display Fields 5-14
Literal Characters in Forms 5-15
TABLES Section 5-18
Table Aliases 5-19
ATTRIBUTES Section 5-20
Fields Linked to Database Columns 5-21
FORMONLY Fields 5-24
Multiple-Segment Fields 5-26
Field Attributes 5-27
Field Attribute Syntax 5-28
AUTONEXT 5-30
COLOR 5-31
COMMENTS 5-36
DEFAULT 5-38
DISPLAY LIKE 5-40
DOWNSHIFT 5-41
FORMAT 5-42

INCLUDE 5-44
INVISIBLE 5-46
NOENTRY 5-47
PICTURE 5-48
PROGRAM 5-50
REQUIRED 5-52
REVERSE 5-53
UPSHIFT 5-54
VALIDATE LIKE 5-55
VERIFY 5-56
WORDWRAP 5-57
INSTRUCTIONS Section 5-63
Screen Records 5-63
Screen Arrays 5-66
Field Delimiters 5-68
Default Attributes 5-69
Precedence of Field Attribute Specifications 5-72
Default Attributes in an ANSI-Compliant Database 5-72
Creating and Compiling a Form 5-73
Compiling a Form Through the Programmer's Environment 5-73
Compiling a Form Through the Operating System 5-74
Default Forms 5-75
Using PERFORM Forms in 4GL 5-77

Chapter 6 **INFORMIX-4GL Reports**

Output from 4GL Programs 6-3
Features of 4GL Reports 6-3
Producing 4GL Reports 6-4
The Report Driver 6-5
The REPORT Definition 6-5
DEFINE Section 6-8
OUTPUT Section 6-9
ORDER BY Section 6-18
FORMAT Section 6-23
FORMAT Section Control Blocks 6-27
AFTER GROUP OF 6-29
BEFORE GROUP OF 6-31
FIRST PAGE HEADER 6-33
ON EVERY ROW 6-34
ON LAST ROW 6-36
PAGE HEADER 6-37
PAGE TRAILER 6-38

Statements in REPORT Control Blocks 6-39
NEED 6-40
PAUSE 6-41
PRINT 6-42
SKIP 6-52

Appendix A **The Demonstration Database and Application**

Appendix B **INFORMIX-4GL Utility Programs**

Appendix C **Using C with INFORMIX-4GL**

Appendix D **Environment Variables**

Appendix E **Native Language Support Within INFORMIX-4GL**

Appendix F **Modifying termcap and terminfo**

Appendix G **The ASCII Character Set**

Appendix H **Reserved Words**

Appendix I **Developing Applications for International Markets**

Glossary

Index

Introduction

Documentation Included with 4GL 4

Other Useful Documentation 5

Conventions of this Manual 5
- Typographical Conventions 5
- Syntax Notation 6

Useful On-Line Files 10

On-Line Error Messages 11

The stores Demonstration Application and Database 12

New Features in 4GL 12
- NLS Support 12
- Improved Performance 12
- Improved Quality 13

Accessing Databases from Within 4GL 13
- Preparing SQL Statements 13

Compatibility and Migration 14

Chapter Overview

INFORMIX-4GL, often called **4GL** in this manual, is a high-level programming language for creating relational database management system (RDBMS) applications in a UNIX environment. As Chapter 3 explains, **4GL** is a superset of the industry-standard SQL structured query language. This book describes the **INFORMIX-4GL** language, including the syntax of **4GL** statements, functions, forms, reports, and operators.

By using **INFORMIX-4GL**, you can efficiently produce complex interactive database applications for data entry, data retrieval and display, and report generation. **4GL** provides all the tools necessary to create screen forms, construct and manage program modules, debug programs, and compile source modules.

Documentation Included with 4GL

The **INFORMIX-4GL** documentation set includes the following manuals:

Manual	Description
INFORMIX-4GL Concepts and Use	Introduces **4GL** and provides the context needed to understand the other manuals in the documentation set. It covers **4GL** goals (what kinds of programming the language is meant to facilitate), concepts and nomenclature (parts of a program, ideas of database access, screen form, and report generation), and methods (how groups of language features are used together to achieve particular effects).
INFORMIX-4GL Reference	The day-to-day, keyboard-side companion for the **4GL** programmer. It describes the features and syntax of the **4GL** language, including **4GL** statements, forms, reports, and the built-in functions and operators. Appendixes are included that describe the demonstration database, the application programming interface of **4GL** with the C language, and utility programs such as **mkmessage** and **upscol**, among other topics.
INFORMIX-4GL by Example	A collection of 30 annotated **4GL** programs. Each is introduced with an overview; then the program source code is shown with line-by-line notes. The program source files are distributed as text files with the product; scripts that create the demonstration database and copy the applications are also included.
INFORMIX-4GL Quick Syntax	Contains the syntax diagrams from the *INFORMIX-4GL Reference*, the *Guide to the Guide to the INFORMIX-4GL Interactive Debugger*, and the *Informix Guide to SQL: Syntax*.
Informix Guide to SQL: Tutorial	Provides a tutorial on SQL as it is implemented by Informix products, and describes the fundamental ideas and terminology that are used when planning and implementing a relational database. It also describes how to retrieve information from a database, and how to modify a database.
Informix Guide to SQL: Reference	Provides full information on the structure and contents of the demonstration database that is provided with **4GL**. It includes details of the Informix system catalog tables, describes Informix and common environment variables that should be set, and describes the column data types that are supported by Informix database engines. It also provides a detailed description of all of the SQL statements that Informix products support.
Informix Guide to SQL: Syntax	Contains syntax diagrams for all of the SQL statements and statement segments that are supported by the 6.0 server. However, not all the statements and segments described in the *Informix Guide to SQL: Syntax* are directly supported by **4GL**. Syntax introduced after Version 5.0 of the server can only be used if it is prepared. For information on preparing statements in **4GL** programs, see page 3-245.
Informix Error Messages, Version 6.0	Lists all the error messages that can be generated by the different Informix products. This document is organized by error message number; it lists each error message and describes the situation that causes the error to occur.

Other Useful Documentation

Depending on the database server that you are using, you or your system administrator need either the *INFORMIX-OnLine Dynamic Server Administrator's Guide*, Version 6.0 or the *INFORMIX-SE Administrator's Guide*, Version 6.0.

Conventions of this Manual

This manual assumes that you are using **INFORMIX-OnLine** as your database server.

The terms "you," "programmer," and "developer" are synonyms for the person (or code-generating program) that writes a **4GL** program. These contrast with the term "user," which refers to the end-user of a **4GL** application.

Informix supports two versions of **4GL**. Both versions use the same **4GL** statements; the versions differ in how they compile and execute code. In this document, **INFORMIX-4GL** or **4GL** refers to the **C Compiler Version** of **4GL**. **Rapid Development System Version** or **RDS** refers to the **Rapid Development System Version** of **4GL**. Chapter 1, "Compiling INFORMIX-4GL Source Files," describes the differences between the two versions of **4GL** and explains how to use both versions.

Sections that follow describe conventions that are used in this manual for typographical format and syntax diagrams.

Typographical Conventions

Informix product manuals use a standard set of conventions to introduce new terms, illustrate screen displays, describe command syntax, and so forth. The following typographical conventions are used throughout this manual:

KEYWORD	All keywords appear in UPPERCASE letters. (You can in fact enter keywords in either uppercase or lowercase letters.)
italics	New terms and emphasized words are printed in *italics*. *Italics* also mark syntax terms for which you must specify some valid identifier, expression, keyword, or statement.
boldface	**4GL** identifiers, SQL identifiers, filenames, database names, table names, column names, utilities, command-line specifications, and similar names are printed in **boldface**.
`monospace`	Output from **4GL**, code examples, and information that you or the user enters are printed `in this typeface.`

. . . Ellipses (. . .) in examples of code or of output mean that text has been omitted to save space or to simplify an illustration. These symbols only appear *within* an example; they are not shown at the beginning or the end of a program fragment. (Ellipses can also appear in ring menus to indicate additional menu options.)

Syntax Notation

SQL statement syntax is described in the *Informix Guide to SQL: Syntax*. The syntax of other **4GL** statements is described in Chapter 3 of this manual.

Syntax diagram conventions are described in this section. Each diagram displays the sequences of required and optional keywords, terms, and symbols that are valid in a given **4GL** statement, command line, or other specification, as in the following diagram of the OPEN FORM statement of **4GL**.

```
OPEN FORM form FROM "filename" ----------------------------------|
```

The following are the three most important rules to remember regarding terms that appear in the syntax diagrams of this book:

- For ease of identification, all **4GL** keywords (like OPEN) are shown in UPPERCASE characters, even though you can enter them in lowercase.
- Terms for which you must supply specific values or names are in *italics*. In this example, *form* and *filename* must be replaced by identifiers.
- All punctuation and other non-alphabetic characters are literal symbols. In this example, the quotation marks (**"**) are literal symbols.

Each syntax diagram begins at the upper left, and ends at the upper right with a vertical terminator. Between these points, any path that does not stop or reverse direction describes a possible form of the statement. (For a few diagrams, notes in the text identify path segments that are mutually exclusive.)

Syntax elements in a path represent terms, keywords, symbols, and segments that can appear in your statement. Except for separators in loops (page Intro-8), which the path approaches counter-clockwise from the right, the path always approaches elements from the left, and continues to the right. Unless otherwise noted, at least one blank character separates syntax elements.

You may encounter one or more of the following elements on a path:

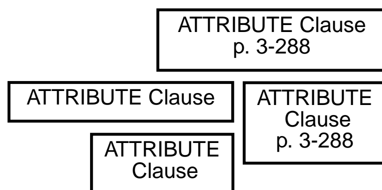
KEYWORD — Spell any word in UPPERCASE letters exactly as shown; you can, however, type it in either uppercase or lowercase letters.

(. , ; @ + * - /) — Punctuation and other *non-alphanumeric* characters are literal symbols that you must enter exactly as shown.

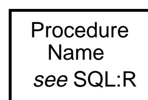
" " — *Double quotes* must be entered as shown. If you prefer, you can replace the pair of double quotes with a pair of

' ' — *single quotes*, but you cannot mix double and single quotes.

variable — A word in *italics* represents a term that you must supply. An explanation below the diagram identifies what values, identifiers, or keywords you can substitute for the italicized term.



A term in a *rectangle* represents a subdiagram on the same page (if no page number is supplied) or on a specified page, as if the subdiagram were spliced into the diagram at this point. (Here “segment” and “subdiagram” are synonyms.) The aspect ratio is not significant. That is, the same segment can be represented by rectangles of different shapes, as in these symbols for the ATTRIBUTE Clause segment.



A reference to SQL:R in a syntax diagram represents an SQL statement or segment that is described in the *Informix Guide to SQL: Reference*. Imagine that the segment were spliced into the diagram at this point.

An *icon* is a warning that this path is valid only for some products, or only under certain conditions. Symbols on the icons indicate what products or conditions support the path.

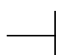
These icons that appear in the *Informix Guide to SQL: Reference* can also appear in a **4GL** syntax diagram:

This path is valid only for **INFORMIX-SE**.

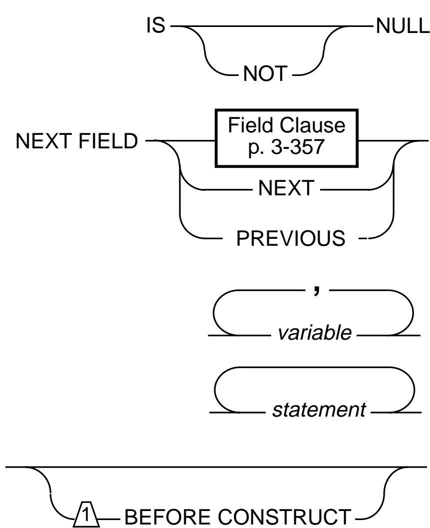
This path is valid only for **INFORMIX-OnLine**.

A *shaded* option is the default. If you do not specify any of the available options, then by default, this option is in effect.

Syntax enclosed between a pair of arrows is a *subdiagram*.



The vertical line is a *terminator*. This only appears at the right, indicating that the syntax diagram is complete.



A *branch* below the main path indicates an optional path. (Any term on the main path is required, unless a branch can circumvent it.)

A set of *multiple branches* indicates a syntax context where a choice among more than two different paths is available.

A *loop* indicates a path that can be repeated. Punctuation along the top of the loop indicates the *separator symbol* for list items, as in this example. If no symbol appears, a *blank space* is the separator, or (as here) the Linefeed that separates each successive **4GL** *statement* within a source module.

A *gate* (1) on a path indicates that you can only use that path the indicated number of times, even if it is part of a larger loop. Here BEFORE CONSTRUCT can be specified no more than once within this **4GL** statement segment.

Icons that appear in the left margin of the text indicate that the accompanying shaded text is valid only for some products or under certain conditions. In addition to the icons described in the preceding list, you may encounter the following icon in the left margin:

ANSI	This icon indicates that the functionality described in the shaded text is valid only if your database is ANSI-compliant.
NLS	This icon indicates that the functionality described in the shaded text is valid only if you are using NLS.

The grey labels and arrows in the following illustration identify the elements of a syntax diagram for the INITIALIZE statement of **4GL**.

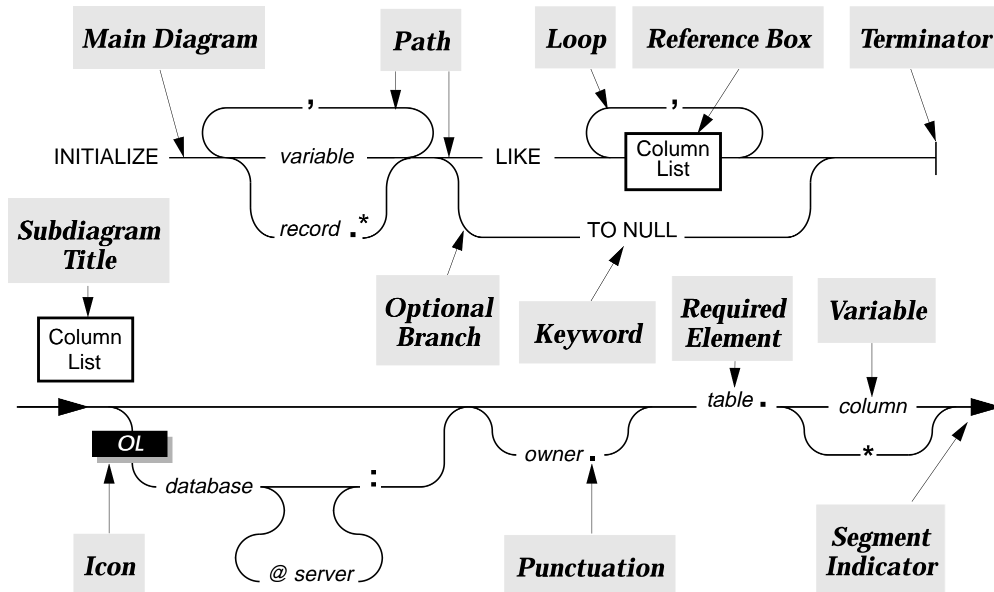


Figure 1 ***Elements of a syntax diagram***

To construct a statement using this diagram, start at the top left with the keyword INITIALIZE. Then follow the diagram to the right, proceeding through the options that you want. The diagram conveys the following information:

1. You must type the keyword INITIALIZE.
2. You must supply the name of a **4GL** *variable* or *record*.
3. You can repeat step (2), using the comma (**,**) symbol to separate *name*s.
4. If you want to assign NULL values to the variables, follow the lower branch of the diagram and type the TO NULL keywords. Now the path leads to the terminator, and your INITIALIZE statement is complete.
5. The alternative to (4) is to assign the *default values* of database columns (from the **syscolval** table) to the simple variables and record member variables that you specified in (2). To do this, type the LIKE keyword, and then follow the subdiagram that describes the "Column List" segment to specify a list of one or more database *columns*.
   - If you are using **INFORMIX-OnLine**, you have the option of specifying the name of a *database*. If you do, you can also specify the name of

a server, prefixed by the `@` symbol. You must follow the *database* or the *database@server* qualifier with a colon (`:`) symbol.

- Regardless of your engine, you have the option of including the *owner* in the qualifier of the column. (This is diagrammed as an option, even though it is sometimes required, for example, in an ANSI-compliant database, if the user who runs the program is not the owner of *table*.) You must follow the *owner* qualifier with a period (`.`) symbol.
- You must type the name of a *table*.You must follow this *table* qualifier with a period (`.`) symbol.
- You can type the name of a *column* in *table*; otherwise, you must type an asterisk (`*`) symbol to specify all the columns in *table*.
- You have reached the segment indicator at the end of the subdiagram, so you must return to ''Column List'' box in the main diagram.

6. Once you are back at the main diagram, you have two options:
   - Type a comma (`,`) and loop back to the subdiagram to type another *table*`.*` or *table*`.`*column* specification, just as you did in step (5).
   - Alternatively, you can continue to the terminator. When you reach the terminator, the INITIALIZE statement is complete.

Note: *When you are instructed to "enter" characters or to "execute" a command, immediately press* ENTER *or* RETURN *after the entry. When you are instructed to "type" the text or to "press" other keys, no* ENTER *nor* RETURN *is required.*

Useful On-Line Files

In addition to the Informix set of manuals, the following on-line files, located in the **$INFORMIXDIR/release** directory, may supplement the information in this manual:

Documentation Notes describe features not covered in the manuals or that have been modified since publication. The file containing the documentation notes for **4GL** is called **4GLDOC_6.0**.

Release Notes describe feature differences from earlier versions of Informix products and how these differences may affect current products. The file containing the release notes for this product is called **TOOLS_6.0**.

Please examine these files because they contain important information about application and performance issues.

4GL provides on-line Help; invoke Help by pressing CONTROL-W.

On-Line Error Messages

Use the **finderr** script to display a particular error message or messages on your terminal screen. The script is located in the **$INFORMIXDIR/bin** directory.

The finderr script has the following syntax:

```
finderr ──────────────┬─ msg_num ─┬──────────┤
                      └───────────┘
```

msg_num Indicates the number of the error message to display. Error messages range from -1 to -32000. Specifying the - sign is optional.

You can specify up to 16 error messages per **finderr** command. **finderr** copies all the specified messages to standard output.

For example, to display the -359 error message, you can enter either of the following:

```
finderr -359
```

or, equivalently:

```
finderr 359
```

The following example demonstrates how to specify a list of error messages. The example also pipes the output to the UNIX **more** command to control the display. You can also direct the output to another file so that you can save or print the error messages:

```
finderr 233 107 113 134 143 144 154 | more
```

A few messages have positive numbers. These messages are used solely within the application tools. In the unlikely event that you want to display them, you must precede the message number with the + sign.

The messages numbered -1 to -100 can be platform-dependent. If the message text for a message in this range does not apply to your platform, check the operating system's documentation for the precise meaning of the message number.

The stores Demonstration Application and Database

4GL includes several **4GL** demonstration applications, along with a demonstration database called **stores2** that contains information about a fictitious wholesale sporting-goods distributor. You can create the **stores2** database in the current directory by entering one of the following commands.

- If you are using the **INFORMIX-4GL C Compiler Version**, type:

  ```
  i4gldemo
  ```

- If you are using the **Rapid Development System Version**, type:

  ```
  r4gldemo
  ```

Many (but not all) of the examples in the **4GL** documentation set are based on the **stores2** database. This database is described in detail in Appendix A.

New Features in 4GL

This version of **4GL** provides support for developers working in European countries, improved performance, and improved quality.

NLS Support

Native Language Support (NLS) is supplied to meet the needs of European countries. This feature extends the ASCII character set from 128 to 256 characters. These additional characters allow you to include characters such as Ö and ç in the definition of your database and in **4GL** programs. NLS provides character sorting and comparison specific to particular languages, and region-specific monetary and numeric information. To use NLS, you need to set some environment variables. You must set the environment variables to the same values as the variables are set for the database (as set by the database creator). NLS is described in detail in Appendix E.

Improved Performance

The removal of the relay module in the 6.0 engine results in improved speed in which data can be retrieved from and sent to a database. As a result, the performance of your **4GL** applications that access a database should improve.

Improved Quality

Over 200 bug fixes have been made to this version of the product. Also, the documentation set has been completely reorganized, rewritten, and updated to include all 6.0 **4GL** features.

Accessing Databases from Within 4GL

Version 6.0 of **4GL** can access any 6.0 Informix server. This includes both **INFORMIX-OnLine** and **INFORMIX-SE**. It cannot, however, access older versions of the server, such as version 5.0.

You access a database in a **4GL** program by placing SQL statements in the program. The version 6.0 **4GL** compiler does not recognize some SQL statements. To include these statements in your **4GL** program, you must prepare these statements so that the compiler knows to pass them on to the engine for processing.

This section describes how to prepare statements. It also lists the SQL statements that are unrecognized. Note that some SQL statements are only unrecognizable if you specify certain options.

For the syntax of the SQL statements allowed in a **4GL** program, see the *INFORMIX-4GL Quick Syntax* guide. This document identifies (with a 6.0 icon) the statements that need to be prepared. For additional information on SQL statements, see the 6.0 server documentation, including the *Informix Guide to SQL: Syntax*, or *Informix Guide to SQL: Reference*, Version 6.0.

Preparing SQL Statements

You prepare SQL statements that the **4GL** compiler will not recognize by including the SQL statement within a PREPARE statement. For example, you can type the following:

```
PREPARE new_procedure FROM
    "CREATE PROCEDURE FROM "/usr/dev/elke/my_procedure""
```

All SQL statements introduced by the 5.0 server and later are not recognized within version 6.0 of **4GL** and therefore must be prepared. For information on using the PREPARE statement, see page 3-245. For a list of statements that must be prepared, see "Preparing Statements in 4GL" on page 3-247.

Compatibility and Migration

You can easily use applications developed with an earlier version of **4GL**, such as Version 4.0 or 4.1, with this 6.0 version of **4GL**. Also, if you have **4GL** applications written for the Windows environment, you can compile and run the applications in the UNIX environment. For complete information on using a Windows application to the UNIX environment, see the *INFORMIX-4GL Starts Here* manual in the Windows documentation set.

Chapter

1

Compiling INFORMIX-4GL Source Files

Chapter Overview 3

Two Implementations of INFORMIX-4GL 3
- Differences Between the C Compiler and RDS Versions 4
  - Differences in the Programmer's Environment 4
  - Differences in Commands 5
  - Differences in Filename Extensions 5

The C Compiler Version 6
- The Programmer´s Environment 6
  - The INFORMIX-4GL Menu 6
  - The MODULE Design Menu 7
  - The FORM Design Menu 12
  - The PROGRAM Design Menu 16
  - The QUERY LANGUAGE Menu 22
- Creating Programs in the Programmer's Environment 23
  - Creating a New Source Module 23
  - Revising an Existing Module 24
  - Compiling a Source Module 24
  - Linking Program Modules 25
  - Executing a Compiled Program 27

Creating Programs at the Command Line 27
Creating or Modifying a 4GL Source File 29
Compiling a 4GL Module 29
Compiling and Linking Multiple Source Files 29
Running 4GL Programs 32
4GL Programs that Call C Functions 32
Program Filename Extensions 32

The Rapid Development System Version 34
The Programmer´s Environment 34
The INFORMIX-4GL Menu 34
The MODULE Design Menu 35
The FORM Design Menu 41
The PROGRAM Design Menu 45
The QUERY LANGUAGE Menu 51
Creating Programs in the Programmer's Environment 52
Creating a New Source Module 52
Revising an Existing Module 53
Compiling a Source Module 53
Combining Program Modules 54
Executing a Compiled RDS Program 56
Invoking the Debugger 56
Creating Programs at the Command Line 56
Creating or Modifying a 4GL Source File 58
Compiling an RDS Source File 58
Concatenating Multi-Module Programs 60
Running RDS Programs 62
Running Multi-Module Programs 63
Running Programs with the Interactive Debugger 63
RDS Programs that Call C Functions 64
Editing the *fgiusr.c* File 65
Creating a Customized Runner 67
Running Programs that Call C Functions 70
Program Filename Extensions 71

Chapter Overview

This chapter describes how to create **INFORMIX-4GL** source-code modules, and how to produce executable **4GL** programs from these source-code modules, both at the operating system prompt and from within the INFORMIX-4GL Programmer's Environment.

The procedures to do this are described for the **INFORMIX-4GL C Compiler Version**, as well as for the **INFORMIX-4GL Rapid Development System**. These two implementations of **4GL** differ in how they process **4GL** source-code modules.

This chapter begins by describing the differences between the two implementations of **4GL**. It then goes on to describe each implementation of **4GL**. The **INFORMIX-4GL C Compiler Version** is described first, beginning on page 1-6. The description of the **INFORMIX-4GL Rapid Development System** begins on page 1-34.

Except as otherwise noted, the other chapters and appendixes of this manual describe features that are identical in both the **C Compiler Version** and **Rapid Development System Version** implementations of **INFORMIX-4GL**.

Two Implementations of INFORMIX-4GL

To write an **INFORMIX-4GL** program, you must first create an ASCII file of **4GL** statements that perform logical tasks to support your application. Other chapters and appendixes describe the features of the **4GL** application development language, and the use and syntax of its statements and utilities. This chapter explains the procedures by which you can transform one or more source-code files of **4GL** statements into an executable **4GL** program.

Informix Software, Inc., offers two different implementations of the **4GL** application development language:

- The **INFORMIX-4GL** *C Compiler Version*, which uses a preprocessor to generate **INFORMIX-ESQL/C** source code. This code is preprocessed

in turn to produce *C source code*, which is then compiled and linked as object code in an executable command file.

- The **INFORMIX-4GL** *Rapid Development System*, which uses a compiler to produce *pseudo-code* (called "p-code") in a single step. You then invoke a "runner" to execute the p-code version of your application. (The **INFORMIX-4GL Rapid Development System** is sometimes abbreviated as **RDS**.)

Differences Between the C Compiler and RDS Versions

Both implementations of **INFORMIX-4GL** use the same **4GL** statements, and nearly identical Programmer's Environments. Because they use different methods to compile your **4GL** source files into executable programs, however, there are a few differences in the user interfaces.

Differences in the Programmer's Environment

The Programmer's Environment is a system of menus that supports the various steps in the process of developing **4GL** application programs. The **Drop** option on the PROGRAM Design menu of the **C Compiler Version** is called **Undefine** in the **INFORMIX-4GL Rapid Development System** implementation.

The **New** and **Modify** options of the PROGRAM Design menu display a different screen form in the two implementations. Both of these screen forms are illustrated later in this chapter.

The **INFORMIX-4GL Rapid Development System** includes a **Debug** option on its MODULE Design menu and PROGRAM Design menu. This option does not appear in the **C Compiler Version**. (The **Debugger** is based on p-code, so it can execute programs and modules compiled by the **INFORMIX-4GL Rapid Development System**.)

The **INFORMIX-4GL Interactive Debugger** is available as a separate product.

Differences in Commands

The commands you use to enter the Programmer's Environments, compile and execute **4GL** programs, and build or restore the stores demonstration database vary between implementations of **4GL**.

C Compiler	RDS	Effect of Command
`i4gl`	`r4gl`	Enter Programmer's Environment
`c4gl` *sfile*`.4gl`	`fglpc` *sfile*	Compile **4GL** source file *sfile*.4gl
xfile`.4ge`	`fglgo` *xfile*	Execute compiled **4GL** program *xfile*
`i4gldemo`	`r4gldemo`	Create the demonstration database

The **INFORMIX-4GL C Compiler Version** requires no equivalent command to the `fglgo` command, since its compiled object files are executable without a runner. The **INFORMIX-4GL Rapid Development System** also contains a command-file script to compile and execute **4GL** programs that call C functions or **INFORMIX-ESQL/C** functions, as described on page 1-64.

Differences in Filename Extensions

The differences in filename extensions are as follows:

C Compiler	RDS	Significance of Extension
.o	**.4go**	Compiled **4GL** source-code module
.4ge	**.4gi**	Executable (runable) **4GL** program file

The backup file extensions **.4bo** and **.4be** for compiled modules and programs have the same names in both implementations. These designate files that are not interchangeable between the two **4GL** implementations, however, because object code produced by a C compiler is different from p-code.

Other filename extensions that are the same in both the **C Compiler Version** and **Rapid Development System Version** designate interchangeable files, if you use both implementations of **INFORMIX-4GL** to process the same **4GL** source-code module.

The C Compiler Version

This section describes the **C Compiler Version** of **INFORMIX-4GL**. In particular, this section:

- Identifies and illustrates all the menu options and screen form fields of the Programmer's Environment.
- Describes the steps for compiling and executing **INFORMIX-4GL** programs from the Programmer's Environment.
- Describes the equivalent command-line syntax for compiling and executing **INFORMIX-4GL** programs.
- Identifies the filename extensions of **4GL** source-code, object, error, and backup files.

The Programmer´s Environment

The **INFORMIX-4GL C Compiler Version** provides a series of nested menus, called the *Programmer's Environment*. These menus support the steps of **4GL** program development and keep track of the components of your application. You can invoke the Programmer's Environment by entering `i4gl` at the system prompt.

The INFORMIX-4GL Menu

The `i4gl` command briefly displays the **INFORMIX-4GL** banner. Then a menu appears, called the INFORMIX-4GL menu:

```
INFORMIX-4GL: [Module] Form  Program  Query-language  Exit
Create, modify, or run individual 4GL program modules.

-------------------------------------------------Press CTRL-W for Help------
```

This is the highest menu, from which you can reach any other menu of the Programmer's Environment. You have five options:

Module Work on an **INFORMIX-4GL** program module.

Form Work on a screen form.

Program Specify components of a multi-module program.

Query-language Use the <vk>SQL interactive interface, if you have **INFORMIX-SQL** installed on your system.

Exit Return to the operating system.

The first three options display new menus that are described in the pages that follow. (You can also press CTRL-W at any menu to display an on-line help message that describes your options.) As at any 4GL menu, you can select an option in either of two ways:

- By typing the first letter of the option.
- By using the SPACEBAR or Arrow keys to move the highlight to the option that you choose, and then pressing RETURN.

The MODULE Design Menu

You can press RETURN or type `m` or `M` to select the **Module** option of the INFORMIX-4GL menu. This displays a new menu, called the MODULE Design menu. Use this menu to work on an individual **4GL** source-code module.

```
MODULE: Modify  New  Compile  Program_Compile  Run  Exit
Change an existing 4GL program module.

-------------------------------------------------Press CTRL-W for Help------
```

The MODULE Design menu supports the following options:

Modify Change an existing **4GL** source-code module.

New Create a new source-code module.

Compile Compile a source-code module.

Program_Compile Compile a **4GL** application program.

Run Execute a compiled **4GL** program module or a multi-module application program.

Exit Return to the INFORMIX-4GL menu.

The **Exit** option returns control to the higher menu from which you accessed the current menu.

You can use these options to create and compile source-code modules of a **4GL** application. See "The FORM Design Menu" on page 1-12 for information on creating **4GL** screen forms. For information on how to create and compile programmer-defined help messages for an **INFORMIX-4GL** application, see the description of the **mkmessage** utility in Appendix B.

The *Modify* Option

Select this option to edit an existing **4GL** source-code module. If you select this option, **INFORMIX-4GL** requests the name of the **4GL** source-code file to be modified and then prompts you to specify a text editor. If you have designated an editor with the DBEDIT environment variable (see Appendix D) or named an editor previously in this session at the Programmer's Environment, **INFORMIX-4GL** invokes that editor. The **4GL** source file whose filename you specified is the **current file**.

When you leave the editor, **INFORMIX-4GL** displays the MODIFY MODULE menu, with the **Compile** option highlighted:

```
MODIFY MODULE:  [Compile]  Save-and-exit  Discard-and-exit
Compile the 4GL module specification.

-------------------------------------------------Press CTRL-W for Help------
```

If you press RETURN or type `c` or `C` to select the **Compile** option, **4GL** displays the COMPILE MODULE menu:

```
COMPILE MODULE:  [Object]  Runable  Exit
Create object file only; no linking to occur.

-------------------------------------------------Press CTRL-W for Help------
```

The **Object** option creates a compiled file with the **.o** extension but makes no attempt to link the file with other files.

The **Runable** option creates a compiled file with the **.4ge** extension. **INFORMIX-4GL** assumes that the current module is a complete **4GL** program, and that no other module needs to be linked to it. Select the **Runable** option if the current program module is a stand-alone **4GL** program. If this is not the case (that is, if the file is one of several **4GL** source-code modules within a multi-module program), then you should use the **Object** option instead, and you must use the PROGRAM Design menu to specify all the component modules.

After you select **Object** or **Runable**, a message near the bottom of the screen will advise you if **INFORMIX-4GL** issues a compile-time warning or error. If there are warnings (but no errors), an executable file is produced. Select the **Exit** option of the next menu, and then **Save-and-exit** at the MODIFY MODULE menu, if you want to save the executable file without reading the warnings.

Alternatively, you can examine the warning messages by selecting **Correct** at the next menu. When you finish editing the **.err** file that contains the warnings, you must select **Compile** again from the MODIFY MODULE menu, since the **Correct** option deletes the executable file.

If there are compilation errors, the following menu appears:

```
COMPILE MODULE: [Correct]  Exit
Correct errors in the 4GL module.

-------------------------------------------------Press CTRL-W for Help------
```

If you choose to correct the errors, an editing session begins on a copy of your source module with embedded error messages. You do not need to delete the error messages, since **INFORMIX-4GL** does this for you. Correct your source file, save your changes, and exit from the editor. The MODIFY MODULE menu reappears, prompting you to recompile, or to save or discard your changes without compiling.

If there are no compilation errors, the MODIFY MODULE menu appears with the **Save-and-Exit** option highlighted. Select this option to save the current source-code module as a file with extension **.4gl**, and create an object file with the same filename, but with the extension **.o**. If you specified **Runable** when

you compiled, the executable version is saved with the extension **.4ge**. The **Discard-and-Exit** option discards any changes to your file since you selected the **Modify** option.

The *New* Option

Select this option to create a new **4GL** source-code module.

```
MODULE: Modify [New] Compile  Program_Compile  Run  Exit
Create a new 4GL program module.

-------------------------------------------------Press CTRL-W for Help------
```

This option resembles the **Modify** option, but NEW MODULE is the menu title, and you must enter a new module name, rather than select it from a list. If you have not designated an editor previously in this session or with DBEDIT, you are prompted for an editor. Then an editing session begins.

The *Compile* Option

The **Compile** option enables you to compile an individual **4GL** source-code module.

```
MODULE: Modify  New [Compile] Program_Compile  Run  Exit
Compile an existing 4GL program module.

-------------------------------------------------Press CTRL-W for Help------
```

After you specify the name of a **4GL** source-code module to compile, the screen displays the COMPILE MODULE menu. Its **Object**, **Runable**, and **Exit** options were described earlier in the discussion of the **Modify** option.

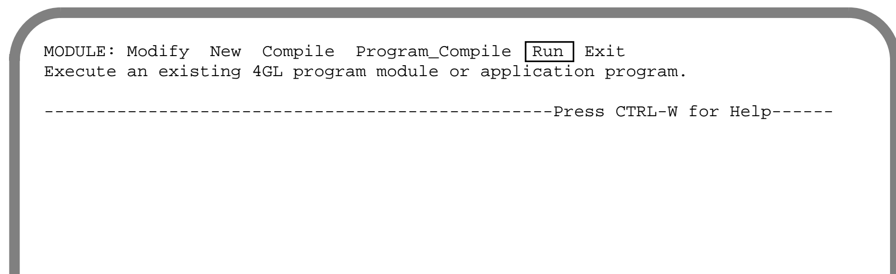
The *Program_Compile* Option

The **Program_Compile** option of the MODULE Design menu is the same as the **Compile** option of the PROGRAM Design menu. (For details, see the previous section, "The Compile Option.") You can use this option to compile and link modules, as described in the program specification database, taking into account the time when the modules were last updated.

This option is useful when you have just modified a single module of a complex program, and need to test it by compiling and linking it with the other modules.

The *Run* Option

Select this option to begin execution of a compiled program.



The RUN PROGRAM screen presents a list of compiled modules and programs, with the highlight on the module corresponding to the current file, if any has been specified. Compiled programs must have the extension **.4ge** to be included in the list. If you compile a program outside the Programmer's Environment and you want it to appear in the program list, give it the extension **.4ge**. If no compiled programs exist, **INFORMIX-4GL** displays an error message and return to the MODULE Design menu.

The *Exit* Option

Select this option to exit from the MODULE Design menu and display the INFORMIX-4GL menu.

```
MODULE: Modify  New  Compile  Program_Compile  Run  [Exit]
Returns to the INFORMIX-4GL menu.

-----------------------------------------------Press CTRL-W for Help------
```

The FORM Design Menu

You can type `f` or `F` at the INFORMIX-4GL menu to select the **Form** option. This option displays a menu, called the FORM Design menu:

```
FORM:  Modify  Generate  New  Compile  Exit
Change an existing form specification.

-----------------------------------------------Press CTRL-W for Help------
```

You can use this menu to create, modify, and compile *screen form* specifications. These define visual displays that **4GL** applications can use to query and modify the information in a database. **INFORMIX-4GL** form specification files are ASCII files that are described in Chapter 5, "Screen Forms."

The FORM Design menu supports the following options:

Modify Change an existing **4GL** screen form specification.

Generate Create a default **4GL** screen form specification.

New Create a new **4GL** screen form specification.

Compile Compile an existing **4GL** screen form specification.

Exit Return to the INFORMIX-4GL menu.

Readers familiar with **INFORMIX-SQL** may notice that this resembles the menu displayed by the **Form** option of the INFORMIX-SQL Main menu.

The *Modify* Option

The **Modify** option of the FORM Design menu enables you to edit an existing form specification file. It resembles the **Modify** option in the MODULE Design menu, since both options are used to edit program modules.

```
FORM: [Modify] Generate  New  Compile  Exit
Change an existing form specification.

-------------------------------------------------Press CTRL-W for Help------
```

If you select this option, you are prompted to select the name of a form specification file to modify. Source files created at the FORM Design menu have the file extension **.per**. (If you use a text editor outside of the Programmer's Environment to create form specification files, you must give them the extension **.per** before you can compile them with the **FORM4GL** screen form facility.)

If you have not already designated a text editor in this **INFORMIX-4GL** session or with DBEDIT, you are prompted for the name of an editor. Then an editing session begins, with the form specification source-code file that you specified as the current file. When you leave the editor, **INFORMIX-4GL** displays the MODIFY FORM menu with the **Compile** option highlighted. Now you can press RETURN to compile the revised form specification file.

```
MODIFY FORM: [Compile]  Save-and-exit  Discard-and-exit
Compile the form specification.

-------------------------------------------------Press CTRL-W for Help------
```

If there are compilation errors, **INFORMIX-4GL** displays the COMPILE FORM menu:

```
COMPILE FORM: [Correct]  Exit
Correct errors in the form specification.

-------------------------------------------------Press CTRL-W for Help------
```

Press RETURN to select **Correct** as your option. An editing session begins on a copy of the current form, with diagnostic error messages embedded where the compiler detected syntax errors. **INFORMIX-4GL** automatically deletes these messages when you save and exit from the editor. After you have corrected the errors, the MODIFY FORM menu appears again, with the **Compile** option highlighted. Press RETURN to recompile. Repeat these steps until the compiler reports no errors.

If there are no compilation errors, you are prompted whether to save the modified form specification file and the compiled form, or to discard the changes. (Discarding the changes restores the version of your form specifications from before you chose the **Modify** option.)

The *Generate* Option

You can type `g` or `G` to select the **Generate** option. This option creates a simple "default" screen form that you can use directly in your program, or that you can later edit by selecting the **Modify** option.

```
FORM:  Modify [Generate]  New  Compile  Exit
Generate and compile a default form specification.

-------------------------------------------------Press CTRL-W for Help------
```

When you select this option, **INFORMIX-4GL** prompts you to select a database, to choose a filename for the form specification, and to identify the tables that the form will access. After you provide these data, **INFORMIX-4GL** creates and compiles a form specification file. (This is equivalent to running the **-d** (default) option of the **form4gl** command, as described in the section titled "Compiling a Form Through the Operating System" on page 5-74.)

The *New* Option

The **New** option of the FORM Design menu enables you to create a new screen form specification.

```
FORM:  Modify  Generate  New  Compile  Exit
Create a new form specification.

-------------------------------------------------Press CTRL-W for Help------
```

After prompting you for the name of your form specification file, **INFORMIX-4GL** places you in the editor where you can create a form specification file. When you leave the editor, **INFORMIX-4GL** transfers you to the NEW FORM menu that is like the MODIFY FORM menu. You can compile your form and correct it in the same way.

The *Compile* Option

The **Compile** option enables you to compile an existing form specification file without going through the **Modify** option.

```
FORM:  Modify  Generate  New  Compile  Exit
Compile an existing form specification.

-------------------------------------------------Press CTRL-W for Help------
```

INFORMIX-4GL compiles the form specification file whose name you specify. If the compilation fails, **INFORMIX-4GL** displays the COMPILE FORM menu with the highlight on the **Correct** option.

The *Exit* Option

The **Exit** option restores the INFORMIX-4GL menu.

```
FORM:  Modify  Generate  New  Compile  [Exit]
Returns to the INFORMIX-4GL menu.

-------------------------------------------------Press CTRL-W for Help------
```

The PROGRAM Design Menu

An **INFORMIX-4GL** program can be a single source-code module that you create and compile at the MODULE Design menu. For applications of greater complexity, however, it is often easier to develop and maintain separate **4GL** modules. The INFORMIX-4GL menu includes the **Program** option so that you can create multi-module programs. If you select this option, **INFORMIX-4GL** searches your DBPATH directories (see Appendix D) for the program specification database, called **syspgm4gl**. This database describes the component modules and function libraries of your **4GL** program.

If **INFORMIX-4GL** cannot find this database, you are asked if you want one created. If you enter `y` in response, **INFORMIX-4GL** creates the **syspgm4gl** database, grants CONNECT privilege to PUBLIC, and displays the PROGRAM Design menu. As Database Administrator of **syspgm4gl**, you can later restrict the access of other users.

If **syspgm4gl** already exists, the PROGRAM Design menu appears.

```
PROGRAM:  [Modify]  New  Compile  Planned_Compile  Run  Drop  Exit
Change the compilation definition of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

You can use this menu to create or modify a multi-module **4GL** program specification, to compile and link a program, or to execute a program.

The PROGRAM Design menu supports the following options:

Modify	Change an existing program specification.
New	Create a new program specification.
Compile	Compile an existing program.
Planned_Compile	List the steps necessary to compile and link an existing program.
Run	Execute an existing program.
Drop	Delete an existing program specification.
Exit	Return to the INFORMIX-4GL menu.

You must first use the MODULE Design menu and FORM Design menu to enter and edit the **INFORMIX-4GL** statements within the component source-code modules of a **4GL** program. Then you can use the PROGRAM Design menu to identify which modules are part of the same application program, and to link all the modules as an executable command file.

The *Modify* Option

The **Modify** option enables you to modify the specification of an existing **4GL** program. (This option is not valid unless at least one program has already been specified. If none has, you can create a program specification by selecting the **New** option from the same menu.) **INFORMIX-4GL** prompts you for the name of the program specification to be modified. It then displays a menu and form that you can use to update the information in the program specification database as shown in Figure 1-1:

```
MODIFY PROGRAM:  4GL  Other  Libraries  Compile_Options  Rename  Exit
Edit the 4GL sources list.

-------------------------------------------------Press CTRL-W for Help------
 Program
 [myprog  ]

 4gl Source    4gl Source Path
 [main      ]  [/u/john/appl/4GL                         ]
 [funct     ]  [/u/john/appl/4GL                         ]
 [rept      ]  [/u/john/appl/4GL                         ]
 [          ]  [                                         ]
 [          ]  [                                         ]

 Other Source  Ext    Other Source Path
 [cfunc     ]  [c  ]  [/u/john/appl/C                           ]
 [          ]  [   ]  [                                         ]
 [          ]  [   ]  [                                         ]
 [          ]  [   ]  [                                         ]

 Libraries [m        ]          Compile Options [          ]
           [         ]                          [          ]
```

Figure 1-1 Example of a Program Specification Entry

The name of the program appears in the Program field. In Figure 1-1 the name is **myprog**. You can change this name by selecting the **Rename** option. **INFORMIX-4GL** assigns the program name, with the extension **.4ge**, to the executable program produced by compiling and linking all the source files and libraries. (Compiling and linking occurs when you select the **Compile** option, as described later in this chapter.) In this example, the resulting executable program would have the name **myprog.4ge**.

Use the **4GL** option to update the entries for the 4gl Source fields and the 4gl Source Path fields on the form. The five rows of fields under these labels form a screen array. When you select the **4GL** option, **INFORMIX-4GL** executes an INPUT ARRAY statement so you can move and scroll through the array. See the INPUT ARRAY statement on page 3-152 for information about how to use

your function keys to scroll, delete rows, and insert new rows. (You cannot redefine the function keys, however, as you can with an **INFORMIX-4GL** program.)

The **INFORMIX-4GL** source program that appears in Figure 1-1 contains three modules:

- One module contains the main program (**main.4gl**).
- One module contains functions (**funct.4gl**).
- One module contains REPORT statements (**rept.4gl**).

Each module is located in the directory **/u/john/appl/4GL**.

If your program includes a module containing only global variables (for example, **global.4gl**), you must also list that module in this section.

Use the **Other** option to include non-**INFORMIX-4GL** source modules or object-code modules in your program. Enter this information into the three-column screen array with the headings Other Source, Ext, and Other Source Path. Enter the filename and location of each non-**INFORMIX-4GL** source-code or object-code module in these fields. Enter the name of the module in the Other Source field, the filename extension of the module (for example, `ec` for an **INFORMIX-ESQL/C** module, or `c` for a C module) in the *Ext* field, and the full directory path of the module in the Other Source Path field. The example in Figure 1-1 includes a file containing C function source-code (**cfunc.c**) located in **/u/john/appl/C**. You can list up to 100 files in this array.

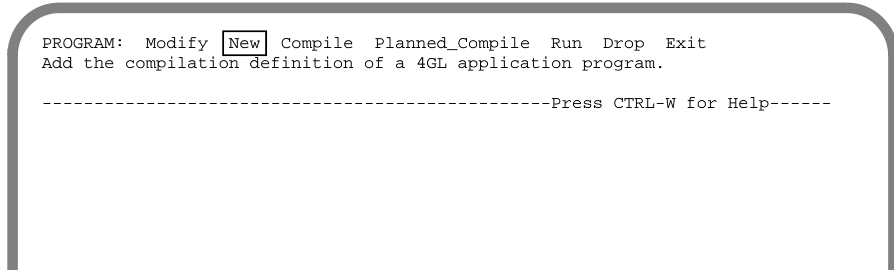
The **Libraries** option enables you to indicate the names of up to ten special libraries to link with your program. **INFORMIX-4GL** calls the C compiler to do the linking and adds the appropriate -**l** prefix, so you should enter only what follows the prefix. The example displayed in Figure 1-1 calls only the standard C math library.

Use the **Compile_Options** option to indicate up to ten C compiler options. Enter this information in the Compile Options field. You cannot, however, specify the -**e** or -**a** options of **c4gl** in this field. (See the section "Creating Programs at the Command Line" on page 1-27 for more information about the options of the **c4gl** command.)

The **Exit** option exits from the MODIFY PROGRAM menu and displays the PROGRAM Design menu.

The *New* Option

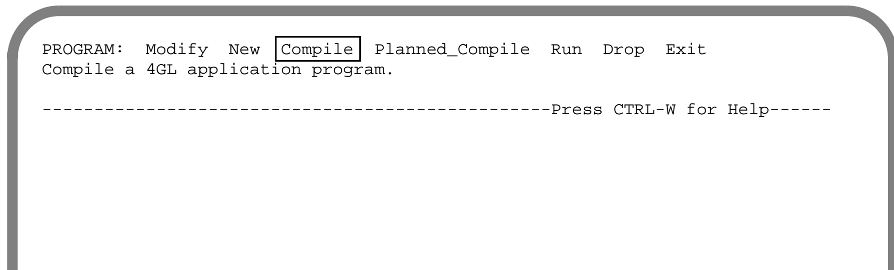
Use the **New** option on the PROGRAM Design menu to create a new specification of the program modules and libraries that make up an application program. You can also specify any necessary compiler or loader options.



The submenu screen forms displayed by the **New** and the **Modify** options of the PROGRAM Design menu are identical, except that you must first supply a name for your program when you select the **New** option. (**INFORMIX-4GL** displays a blank form in the NEW PROGRAM menu.) The NEW PROGRAM menu has the same options as the MODIFY PROGRAM menu, as illustrated earlier.

The *Compile* Option

The **Compile** option performs the compilation and linking described in the program specification database, taking into account the time when each file was last updated. It compiles only those files that have not been compiled since they were changed.



INFORMIX-4GL lists each step of the preprocessing and compilation as it occurs. An example of these messages appears in the illustration of the **Planned_Compile** option, next.

The *Planned_Compile* Option

Taking into account the time when the various files in the dependency relationships last changed, the **Planned_Compile** option prompts for a program name and displays a summary of the steps that will be executed if you select the **Compile** option. No compilation actually takes place.

```
PROGRAM:  Modify  New  Compile  [Planned_Compile]  Run  Drop  Exit
Show the planned compile actions of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
Compiling INFORMIX-4GL sources:
                /u/john/appl/4GL/main.4gl
                /u/john/appl/4GL/funct.4gl
                /u/john/appl/4GL/rept.4gl
Compiling Embedded SQL sources:
Compiling with options:
Linking with libraries:
                m
Compiling/Linking other sources:
                /u/john/appl/C/cfunc.c
```

In this instance, changes were made to all the components of the **4GL** program that were listed in Figure 1-1. This display indicates that no source-code module has been compiled since the program was changed.

The *Run* Option

The **Run** option of the PROGRAM Design menu is the same as the **Run** option of the MODULE Design menu. It displays a list of any compiled programs (files with the extension **.4ge**) and positions the highlight on the current program, if a program has been specified. **INFORMIX-4GL** then executes the program that you select.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  [Run] Drop  Exit
Execute a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

The *Drop* Option

The **Drop** option of the PROGRAM Design menu prompts you for a program name and removes the compilation and linking definition of that program from the **syspgm4gl** database. This action removes the definition only. Your program and **4GL** modules are *not* removed.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  [Drop]  Exit
Drop the compilation definition of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

The *Exit* Option

The **Exit** option clears the PROGRAM Design menu and restores the INFORMIX-4GL menu.

The QUERY LANGUAGE Menu

The <vk>SQL interactive interface is identical to the interactive <vk>SQL interface of **INFORMIX-SQL**. You can use this option only if you have separately purchased **INFORMIX-SQL** and installed it.

The **Query-language** option is placed at the top-level menu so you can test <vk>SQL statements without leaving the **INFORMIX-4GL** Programmer's Environment. You can also use this option to create, execute, and save <vk>SQL scripts.

Creating Programs in the Programmer's Environment

To invoke the **C Compiler Version** of the Programmer's Environment, enter the following command at the system prompt:

```
i4gl
```

After a sign-on message, the INFORMIX-4GL menu appears.

Creating a **4GL** application with the **C Compiler Version** of **INFORMIX-4GL** requires the following steps:

1. Creating a new source module or revising an existing source module.
2. Compiling the source module.
3. Linking the program modules.
4. Executing the compiled program.

This process is described below.

Creating a New Source Module

This section outlines the procedure for creating a new source module. If your source module already exists, see "Revising an Existing Module," next.

1. Select the **Module** option of the INFORMIX-4GL menu by pressing `m` or by pressing RETURN if the **Module** option is highlighted.

   The MODULE Design menu is displayed.
2. If you are creating a new **.4gl** source module, press `n` to select the **New** option of the MODULE Design menu.
3. Enter a name for the new module.

   The name must begin with a letter and can include letters, numbers, and underscores. The name must be unique among the files in the same directory, and among the other program modules, if it will be part of a multi-module program. **INFORMIX-4GL** attaches extension **.4gl** to this identifier, as the filename of your new source module.
4. Press RETURN.

Revising an Existing Module

If you are revising an existing **4GL** source file, follow these steps:

1. Select the **Modify** option of the MODULE Design menu.

   The screen lists the names of all the **.4gl** source modules in the current directory and prompts you to select a source file to edit.

2. Use the Arrow keys to highlight the name of a source module and press RETURN, or enter a filename (with no extension).

   If you specified the name of an editor with the DBEDIT environment variable, an editing session with that editor begins automatically. Otherwise, the screen prompts you to specify a text editor.

   Specify the name of a text editor, or press RETURN for **vi**, the default editor. Now you can begin an editing session by entering **4GL** statements.

3. When you have finished entering or editing your **4GL** code, use an appropriate editor command to save your source file and end the text editing session.

Compiling a Source Module

The **.4gl** source file module that you create or modify is an ASCII file that must be compiled before it can be executed.

1. Select the **Compile** option from the MODULE Design menu.
2. Select the type of module you are compiling, either **Runable** or **Object**.

   If the module is a complete **4GL** program that requires no other modules, select **Runable**. This option first creates an intermediate **ESQL/C** version of your source-code module, then calls the **ESQL/C** preprocessor which produces C output, and finally calls the C compiler to produce a compiled file with the same filename, but with the extension **.4ge**.

   If the module is one module of a multi-module **4GL** program, select **Object**. This option creates a compiled object file module, with the same filename, but with extension **.o**. See also "Linking Program Modules" on page 1-25.

3. If the compiler detects errors, no compiled file is created, and you are prompted to fix the problem.

   Select **Correct** to resume the previous text editing session, with the same **4GL** source code, but with error messages in the file. Edit the file to correct the error, and select **Compile** again. If an error message appears, repeat this process until the module compiles without error.

4. After the module compiles successfully, select **Save-and-exit** from the menu to save the compiled program.

   The MODULE Design menu appears again on your screen.

5. If your program requires *screen forms*, select **Form** from the INFORMIX-4GL menu.

   The FORM Design menu appears. For information about designing and creating screen forms, see Chapter 5.

6. If your program displays help messages, you must create and compile a help file.

   Use the **mkmessage** utility to compile the help file. For more information on this utility, see Appendix B.

Linking Program Modules

If your new or modified module is part of a multi-module **4GL** program, you must link all of the modules into a single program file before you can run the program. If the module that you compiled is the only module in your program, you are now ready to run your program (see "Executing a Compiled Program" on page 1-27).

1. Select the **Program** option from the INFORMIX-4GL menu.

   The PROGRAM Design menu appears.

2. If you are creating a new multi-module **4GL** program, select the **New** option. If you are modifying an existing one, select **Modify**.

   In either case, the screen prompts you for the name of a program.

3. Enter the name (without a file extension) of the program that you are modifying, or the name to be assigned to a new program.

   Names must begin with a letter, and can include letters, underscores (_), and numbers. After you enter a valid name, the PROGRAM screen appears, with your program name in the first field.

   If you selected **Modify**, the names and pathnames of the source-code modules are also displayed. In that case, the PROGRAM screen appears below the MODIFY PROGRAM menu, rather than below the NEW PROGRAM menu. (Both menus list the same options.)

```
MODIFY PROGRAM: [4GL] Other  Libraries  Compile_Options  Rename  Exit
Edit the 4GL sources list.

------------------------------------------------Press CTRL-W for Help------
 Program
 [        ]

 4gl Source    4gl Source Path
 [          ]  [                                      ]
 [          ]  [                                      ]
 [          ]  [                                      ]
 [          ]  [                                      ]
 [          ]  [                                      ]

 Other Source  Ext    Other Source Path
 [          ]  [   ]  [                                      ]
 [          ]  [   ]  [                                      ]
 [          ]  [   ]  [                                      ]
 [          ]  [   ]  [                                      ]

 Libraries [        ]          Compile Options [          ]
```

4. Identify the files that comprise your program:
   - To specify new **4GL** modules or edit the list of **4GL** modules, select the **4GL** option.

     You can enter or edit the name of a module, without the **.4gl** file extension. Repeat this step for every module. If the module is not in the current directory nor in a directory specified by the DBPATH environment variable, enter the pathname to the directory where the module resides.
   - To include any modules in your program that are not **4GL** source files, select the **Other** option.

     This option enables you to specify each filename in the *Other Source* field, the filename extension in the *Ext* field, and the pathname in the *Other Source Path* field.

     These fields are part of an array that can specify up to 100 "other" modules, such as C language source files or object files. If you have the **INFORMIX-ESQL/C** product installed on your system, you can also specify **ESQL/C** source modules (with extension **.ec**) here.
   - To specify any function libraries that should be linked to your program (besides the **INFORMIX-4GL** library that is described in Chapter 4), select the **Libraries** option. This option enables you to enter or edit the list of library names in the *Libraries* fields.
   - To specify compiler flags, select the **Compile_Options** option. These flags can be entered or edited in the *Compile Options* fields.

5. After you have correctly listed all of the modules of your program, select the **Exit** option to return to the PROGRAM Design menu.
6. Select the **Compile** option of the PROGRAM Design menu.

   This option produces an executable file that contains all your **4GL** program modules. Its filename is the program name that you specified, with extension **.4ge**. The screen lists the names of your **.4gl** source modules, and displays the PROGRAM Design menu with the **Run** option highlighted.

Executing a Compiled Program

After compiling and linking your program modules, you can execute your program. To do so, select the **Run** option from the MODULE Design menu. This option begins execution of the compiled **4GL** program.

Your program can display menus, screen forms, windows, or other screen output according to your program logic and any keyboard interaction of the user with the program.

Creating Programs at the Command Line

You can also create **.4gl** source files and compiled **.o** and **.4ge** files at the operating system prompt. Figure 1-2 shows the process of creating, compiling, linking, and running an **INFORMIX-4GL** program from the command line.

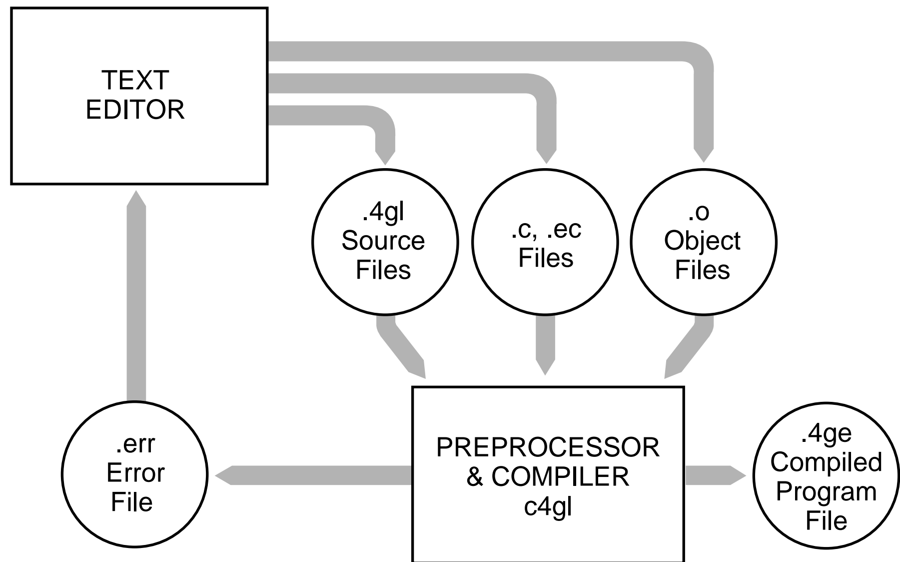


Figure 1-2 ***Creating and Running an*** **INFORMIX-4GL** ***Program***

In Figure 1-2 the rectangles represent processes controlled by specific commands, and the circles represent files. Arrows indicate whether a file can serve as input or output (or as both) for a process.

This diagram is simplified and ignores the similar processes by which forms, help messages, and other components of **4GL** applications are compiled, linked, and executed.

- The cycle begins in the upper left corner with a text editor, such as **vi**, to produce a **4GL** source module.
- A multi-module program can include additional **4GL** source files (**.4gl**), **INFORMIX-ESQL/C** source files (**.ec**), C language source files (**.c**), and object files (**.o**).
- The program module can then be compiled, by invoking the **c4gl** preprocessor and compiler command. (If error messages result, find them in the **.err** file and edit the source file to correct the errors. Then recompile the corrected source module.)

The resulting compiled **.4ge** program file is an executable command file that you can run by entering its name at the system prompt:

```
filename.4ge
```

where *filename*`.4ge` specifies your compiled **4GL** file.

The correspondence between commands and menu options of the Programmer's Environment is summarized by the following list:

Menu Option	Invokes	Command
Module New/Modify	UNIX System Editor	**vi**
Compile	4GL Preprocessor/ C Compiler	**c4gl**
Run	4GL Application	*filename*.**4ge**

Creating or Modifying a 4GL Source File

Use your system editor or another text editing program to create a **.4gl** source file or to modify an existing file. For information on the statements you can include in a **4GL** program, see Chapter 3.

Compiling a 4GL Module

You can compile an **INFORMIX-4GL** source file at the system prompt by entering a command of the form:

```
c4gl source.4gl -o filename.4ge
```

The **c4gl** command compiles your **4GL** source-code module (here called *source*.**4gl**) and produces an executable program called *filename*.**4ge**. The complete syntax of the **c4gl** command appears on the next page.

Compiling and Linking Multiple Source Files

An **INFORMIX-4GL** program can include several source-code modules. You cannot execute a **4GL** program until you have preprocessed and compiled all the source modules and linked them with any function libraries that they reference. You can do all this in a single step at the system prompt by using the **c4gl** command, which performs the following processing steps:

1. Reads your **4GL** source-code files (extension **.4gl**) and preprocesses them to produce **ESQL/C** code.
2. Reads the **ESQL/C** code and preprocesses it to produce C code.
3. Reads the C code and compiles it to produce an object file.

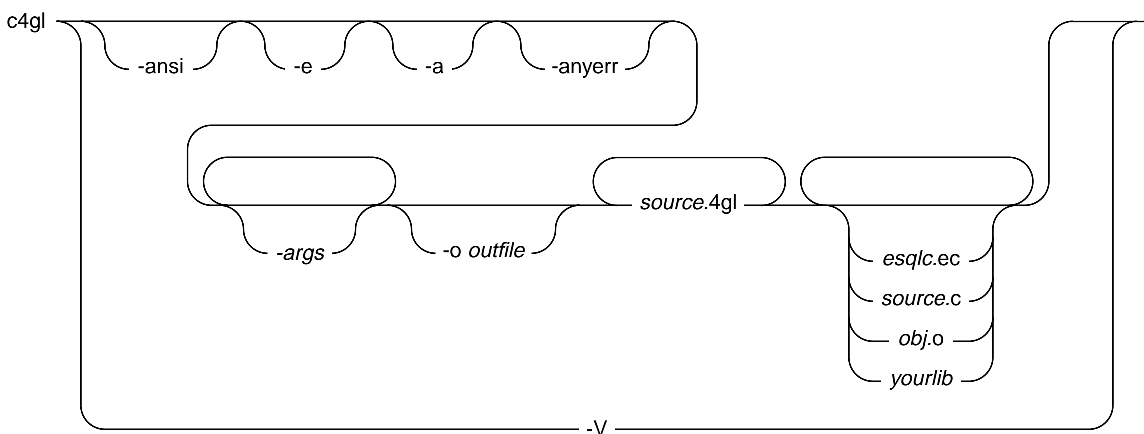
4. Links the object file to the **INFORMIX-ESQL/C** libraries and to any additional libraries that you specify in the command line.

You must assign the filename extension **.4gl** to any **INFORMIX-4GL** source-code modules that you compile. The resulting **.4ge** file is an executable version of your program.

Notice that **ESQL/C** source files (with extension **.ec**), C source files (with extension **.c**), and C object files (with extension **.o**) are intermediate steps in producing an executable **INFORMIX-4GL** program. Besides **4GL** source files (with extension **.4gl**), you can also include files of any or all of these types when you specify a **c4gl** command line to compile and link the component modules of a **4GL** program.

c4gl Command

The **c4gl** command supports the following syntax:



-args	are other arguments for your C compiler.
esql.**ec**	is an **ESQL/C** source file to compile and link.
obj.**o**	is an object file to link with your **4GL** program.
outfile	is a name that you assign to the compiled **4GL** program.
source.**4gl**	is the name of an **4GL** source module. You must specify the **.4gl** extension.
src.**c**	is a C language source file to compile and link.
yourlib	is a library from which to extract functions that are not part of the **4GL** or **ESQL/C** libraries.

Usage

The **c4gl** command passes all C compiler arguments (*args*) and other C source and object files (*src*.**c**, *obj*.**o**) directly to the C compiler (**cc**).

You can compile **4GL** modules separately from your MAIN program block. If there is no MAIN program block in *source*.**4gl**, your code is compiled to *source*.**o**, but not linked with other modules or libraries. You can use **c4gl** to link your code with a module that includes the MAIN program block at another time. (For more information, see the description of the MAIN statement on page 3-191.)

To have your compiled program check array bounds at run time, include the -**a** option. As shown in the syntax diagram, the -**a** option must appear on the command line before the *source*.**4gl** filename. The -**a** option requires additional run-time processing, so you may want to use this option only during development to debug your program.

To instruct the compiler to check all <vk>SQL statements for compliance with ANSI standards, include the -**ansi** option. If you specify the -**ansi** option, it must appear first in your list of **c4gl** command arguments. The -**ansi** option asks for compile-time warning messages if your source code includes Informix extensions to the ANSI standard for <vk>SQL. Compiler warnings and error messages are saved in a file called *source*.**err**.

To perform only the preprocessor steps, with no compilation or linking, include the -**e** option.

If you specify the -**anyerr** option, 4GL sets the **status** variable after evaluating expressions. The -**anyerr** option overrides any WHENEVER ERROR statements in your program.

If you omit the -**o** *outfile* option, the default filename is **a.out**.

To display the release version number of your <vk>SQL software, use the -**V** option. If you specify the -**V** option, all other arguments are ignored, and no output files are produced.

Examples

The simplest case is to compile a single-module **INFORMIX-4GL** program. This command produces an executable program called **single.4ge**:

```
c4gl single.4gl -o single.4ge
```

In the next example, the object files **mod1.o**, **mod2.o**, and **mod3.o** are previously compiled **INFORMIX-4GL** modules, and **mod4.4gl** is a source-code module. Suppose that you want to compile and link **mod4.4gl** with the three object modules to create an executable program called **myappl.4ge**. To do so, enter the following command line:

```
c4gl mod1.o mod2.o mod3.o mod4.4gl -o myappl.4ge
```

Running 4GL Programs

As noted in the previous section, a valid **c4gl** command line produces a **.4ge** file (or whatever you specify after the **-o** argument) that is an executable command file.

To execute your compiled **INFORMIX-4GL** application program, enter the filename at the system prompt. For example, to run **myappl.4ge** (the program in the previous example), simply enter the command line:

```
myappl.4ge
```

Some **INFORMIX-4GL** programs may require additional command-line arguments, such as constants or filenames, depending on the logic of your specific **4GL** application.

4GL Programs that Call C Functions

No special procedures are needed to create, compile, and execute **4GL** programs that call C functions or **INFORMIX-ESQL/C** functions when you use the **C Compiler Version** of **INFORMIX-4GL**. For information on creating **INFORMIX-4GL** programs that call programmer-defined C functions within **4GL** modules, see Appendix C.

Program Filename Extensions

Source, runable, error, and backup files generated by **INFORMIX-4GL** are stored in the current directory and are labeled with a filename extension. The following list shows the file extensions for the source, runable, and error files. These files are produced during the normal course of using the **C Compiler Version** of **INFORMIX-4GL**.

File	Description
file.4gl	**4GL** source file.
file.o	**4GL** object file.
file.4ge	**4GL** executable (runable) file.

File	Description
file.err	**4GL** source error file, created when an attempt to compile a module fails. The file contains **4GL** source code, plus any compiler syntax error or warning messages.
file.ec	Intermediate source file, created during the normal course of compiling an **4GL** module.
file.c	Intermediate C file, created during the normal course of compiling an **4GL** module.
file.erc	**4GL** object error file, created when an attempt to compile or to link a non-**4GL** source-code or object module fails. The file contains **4GL** source code and annotated compiler errors.
form.per	**FORM4GL** source file.
form.frm	**FORM4GL** object file.
form.err	**FORM4GL** source error file.

The last three files do not exist unless you create or modify a screen form specification file, as described in Chapter 5.

The following list identifies the backup files that are produced when you use **INFORMIX-4GL** from the Programmer's Environment:

File	Description
file.4bl	**4GL** source backup file, created during the modification and compilation of a **.4gl** program module.
file.4bo	Object backup file, created during the compilation of a **.o** program module.
file.4be	Object backup file, created during the compilation of a **.4ge** program module.
file.pbr	**FORM4GL** source backup file.
file.fbm	**FORM4GL** object backup file.

Under normal conditions, **INFORMIX-4GL** creates the backup files and intermediate files as necessary and deletes them upon a successful compilation. If you interrupt a compilation, you may find one or more of these files in your current directory.

During the compilation process, **INFORMIX-4GL** stores a backup copy of the **file.4gl** source file in **file.4bl**. The time stamp is modified on the (original) **file.4gl** source file, but not on the backup **file.4bl** file. In the event of a system crash, you may need to replace the modified **file.4gl** file with the backup copy contained in the **file.4bl** file.

The Programmer's Environment does not allow you to begin modifying a **.4gl** or **.per** source file if the corresponding backup file already exists in the same directory. After an editing session terminates abnormally, for example, you must delete or rename any backup file before you can resume editing your **4GL** module or form from the Programmer's Environment.

The Rapid Development System Version

This section describes the **Rapid Development System Version** of **INFORMIX-4GL**. In particular, it:

- Identifies and illustrates all the menu options and screen form fields of the **RDS** Programmer's Environment.
- Describes the steps for compiling and executing **INFORMIX-4GL** programs from the Programmer's Environment.
- Describes the equivalent command-line syntax for compiling and executing **INFORMIX-4GL** programs.
- Identifies the filename extensions of **4GL** source-code, object, error, and backup files.

The Programmer´s Environment

The **INFORMIX-4GL Rapid Development System** provides a series of menus called the *Programmer's Environment.* These menus support the steps of **4GL** program development and keep track of the components of your application. You can invoke the Programmer's Environment by entering `r4gl` at the system prompt.

The INFORMIX-4GL Menu

The **r4gl** command briefly displays the **INFORMIX-4GL** banner and sign-on message. Then a menu appears, called the INFORMIX-4GL menu:

```
INFORMIX-4GL: [Module] Form  Program  Query-language  Exit
Create, modify or run individual 4GL program modules.

-------------------------------------------------Press CTRL-W for Help------
```

This is the highest menu, from which you can reach any other menu of the Programmer's Environment. You have five options:

Module Work on an **INFORMIX-4GL** program module.

Form Work on a screen form.

Program Specify components of a multi-module program.

Query-language Use the <vk>SQL interactive interface, if you have **INFORMIX-SQL** installed on your system.

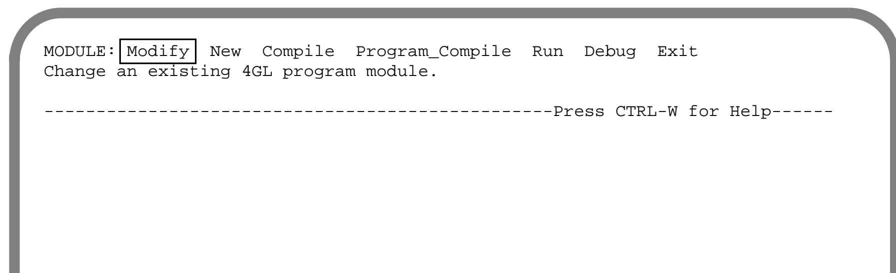
Exit Return to the operating system.

The first three options display new menus that are described in the pages that follow. (You can also press CONTROL-W at any menu to display an on-line help message that describes your options.) As at any 4GL menu, you can select an option in either of two ways:

- By typing the first letter of the option.
- By using the SPACEBAR or Arrow keys to move the highlight to the option that you choose, and then pressing RETURN.

The MODULE Design Menu

You can press RETURN or type m or M to select the **Module** option of the INFORMIX-4GL menu. This option displays a new menu, called the MODULE Design menu. Use this menu to work on an individual **4GL** source-code file.



The MODULE Design menu supports the following options:

Modify Change an existing **4GL** source-code module.

New Create a new **4GL** source-code module.

Compile Compile an existing **4GL** source-code module.

Program_Compile Compile a **4GL** application program.

Run Execute a compiled **4GL** module or multi-module application program.

Debug Invoke the **INFORMIX-4GL Interactive Debugger** to examine an existing **4GL** program module or application program (if you have the **Debugger** product installed on your system).

Exit Return to the INFORMIX-4GL menu.

As in all of the menus of the Programmer's Environment except the INFORMIX-4GL menu, the **Exit** option returns control to the higher menu from which you accessed the current menu.

You can use these options to create and compile source-code modules of a **4GL** application. (For information on creating **4GL** screen forms, see "The FORM Design Menu" on page 1-41. For information on creating and compiling programmer-defined help messages for an **INFORMIX-4GL** application, see the description of the **mkmessage** utility in Appendix B.)

The *Modify* Option

Select this option to edit an existing **4GL** source-code module. You are prompted for the name of the **4GL** source-code file to modify and the text editor to use. If you have designated an editor with the DBEDIT environment variable (see Appendix D) or named an editor previously in this session at the Programmer's Environment, **INFORMIX-4GL** invokes that editor. The **4GL** source file whose filename you specified is the **current file**.

When you leave the editor, **INFORMIX-4GL** displays the MODIFY MODULE menu, with the **Compile** option highlighted:

```
MODIFY MODULE:  Compile   Save-and-exit  Discard-and-exit
Compile the 4GL module specification.

-------------------------------------------------Press CTRL-W for Help------
```

If you press RETURN or type `c` or `C` to select the **Compile** option, **INFORMIX-4GL** displays the COMPILE MODULE menu:

```
COMPILE MODULE:  Object  Runable  Exit
Create object file (.4go suffix).

-------------------------------------------------Press CTRL-W for Help------
```

The **Object** option creates a file with a **.4go** extension. The **Runable** option creates a file with a **.4gi** extension. Select the **Runable** option if the current program module is a stand-alone **4GL** program. If this is not the case, (that is, if the file is one of several **4GL** source-code modules within a multi-module program), then you should use the **Object** option instead, and you must use the PROGRAM Design menu to specify all the component modules.

After you select **Object** or **Runable**, a message near the bottom of the screen will advise you if **INFORMIX-4GL** issues a compile-time warning or error. If there are warnings (but no errors), a p-code file is produced. Select the **Exit** option of the next menu, and then **Save-and-exit** at the MODIFY MODULE menu, if you want to save the p-code file without reading the warnings.

Alternatively, you can examine the warning messages by selecting **Correct** at the next menu. When you finish editing the **.err** file that contains the warnings, you must select **Compile** again from the MODIFY MODULE menu, since the **Correct** option deletes the p-code file.

If there are compilation errors, the following menu appears:

```
COMPILE MODULE:  Correct  Exit
Correct errors in the 4GL module.

-------------------------------------------------Press CTRL-W for Help------
```

If you choose to correct the errors, an editing session begins on a copy of your source module with embedded error messages. (You do not need to delete error messages, since **INFORMIX-4GL** does this for you.) Correct your source file, save your changes, and exit from the editor. The MODIFY MODULE menu reappears, prompting you to recompile, or to save or discard your changes without compiling.

If there are no compilation errors, the MODIFY MODULE menu appears with the **Save-and-Exit** option highlighted. If you select this option, **INFORMIX-4GL** saves the current source-code module as a disk file with the filename extension **.4gl**, and saves the compiled version as a file with the same filename, but with the extension **.4go** or **.4gi**. If you select the **Discard-and-Exit** option, **INFORMIX-4GL** discards any changes made to your file since you selected the **Modify** option.

The *New* Option

Select this option to create a new **4GL** source-code module.

```
MODULE: Modify  New  Compile  Program_Compile  Run  Debug  Exit
Create a new 4GL program module.

-------------------------------------------------Press CTRL-W for Help------
```

This option resembles the **Modify** option, but NEW MODULE is the menu title, and you must enter a new module name, rather than select it from a list. If you have not designated an editor previously in this session or with DBEDIT, you are prompted for an editor. Then an editing session begins.

The *Compile* Option

The **Compile** option enables you to compile an individual **4GL** source-code module without first selecting the **Modify** option.

```
MODULE: Modify  New  [Compile]  Program_Compile  Run  Debug  Exit
Compile an existing 4GL program module.

-------------------------------------------------Press CTRL-W for Help------
```

After you specify the name of a **4GL** source-code module to compile, the screen displays the COMPILE MODULE menu. For information on the COMPILE MODULE menu options, see "The Modify Option" on page 1-36.

The *Program_Compile* Option

The **Program_Compile** option of the MODULE Design menu is the same as the **Compile** option of the PROGRAM Design menu (see "The Compile Option" on page 1-49). It permits you to compile and combine modules as described in the program specification database, taking into account the time when the modules were last updated. This option is useful when you have just modified a single module of a complex program and want to test it by compiling it with the other modules.

The *Run* Option

Select this option to begin execution of a compiled program.

```
MODULE: Modify  New  Compile  Program_Compile  [Run]  Debug  Exit
Execute an existing 4GL program module or application program.

-------------------------------------------------Press CTRL-W for Help------
```

The RUN PROGRAM screen presents a list of compiled modules and programs, with the highlight on the module corresponding to the current file, if any has been specified. Compiled programs must have the extension **.4gi** to be included in the list. If you compile a module with the extension **.4go**, you can run it by typing the filename and extension at the prompt. If no compiled programs exist, **INFORMIX-4GL** displays an error message and restores the MODULE Design menu.

The *Debug* Option

Select this option to use the **INFORMIX-4GL Interactive Debugger** to analyze a program. This option is implemented only if you have separately purchased and installed the **INFORMIX-4GL Interactive Debugger** on your system.

```
MODULE: Modify  New  Compile  Program_Compile  Run  [Debug]  Exit
Returns to the INFORMIX-4GL menu.

-------------------------------------------------Press CTRL-W for Help------
```

If you have the **Debugger** product, refer to the **INFORMIX-4GL Interactive Debugger** documentation for more information about this option.

The *Exit* Option

Select this option to exit from the MODULE Design menu and display the INFORMIX-4GL menu.

```
MODULE: Modify  New  Compile  Program_Compile  Run  Debug  [Exit]
Returns to the INFORMIX-4GL menu.

-------------------------------------------------Press CTRL-W for Help------
```

The FORM Design Menu

You can type `f` or `F` at the INFORMIX-4GL menu to select the **Form** option. This option replaces the INFORMIX-4GL menu with a new menu, called the FORM Design menu:

```
FORM: Modify  Generate  New  Compile  Exit
Change an existing form specification.

-------------------------------------------------Press CTRL-W for Help------
```

You can use this menu to create, modify, and compile *screen form* specifications. These specifications define visual displays that **4GL** applications can use to query and modify the information in a database. **INFORMIX-4GL** screen form specifications are ASCII files that are described in Chapter 5.

The FORM Design menu supports the following options:

Modify Change an existing **4GL** screen form specification.

Generate Create a default **4GL** screen form specification.

New Create a new **4GL** screen form specification.

Compile Compile an existing **4GL** screen form specification.

Exit Return to the INFORMIX-4GL menu.

Readers familiar with the menu system of **INFORMIX-SQL** may notice that this menu resembles the menu displayed by the **Form** option of the INFORMIX-SQL Main menu.

For descriptions of the usage and statement syntax of **4GL** screen form specifications, see Chapter 5.

The *Modify* Option

The **Modify** option of the FORM Design menu enables you to edit an existing form specification file. It resembles the **Modify** option in the MODULE Design menu, since both options are used to edit program modules.

```
FORM:  Modify  Generate  New  Compile  Exit
Change an existing form specification.

-------------------------------------------------Press CTRL-W for Help------
```

If you select this option, you are prompted to select the name of a form specification file to modify. Source files created at the FORM Design menu (or at the command line by the **form4gl** screen form facility) have the file extension **.per**.

If you have not already designated a text editor in this **INFORMIX-4GL** session or with DBEDIT, you are prompted for the name of an editor. Then an editing session begins, with the form specification source-code file that you specified as the current file. When you leave the editor, **INFORMIX-4GL** displays the MODIFY FORM menu with the **Compile** option highlighted.

```
MODIFY FORM:  Compile  Save-and-exit  Discard-and-exit
Compile the form specification.

-------------------------------------------------Press CTRL-W for Help------
```

Now you can press RETURN to compile the revised form specification file. If the compiler finds errors, the COMPILE FORM menu appears:

```
COMPILE FORM:  Correct  Exit
Correct errors in the form specification.

-------------------------------------------------Press CTRL-W for Help------
```

Press RETURN to select **Correct** as your option. An editing session begins on a copy of the current form, with diagnostic error messages embedded where the compiler detected errors. **INFORMIX-4GL** deletes these messages when you save and exit from the editor. After you correct the errors, the MODIFY FORM menu appears again, with the **Compile** option highlighted. Press RETURN to recompile.

If there are no compilation errors, you are prompted whether to save the modified form specification file and the compiled form, or to discard the changes. (Discarding the changes restores the version of your form specifications from before you chose the **Modify** option.)

The *Generate* Option

You can type `g` or `G` to select the **Generate** option. This option creates a simple "default" screen form for use directly in your **4GL** program, or for you to edit later by selecting the **Modify** option.

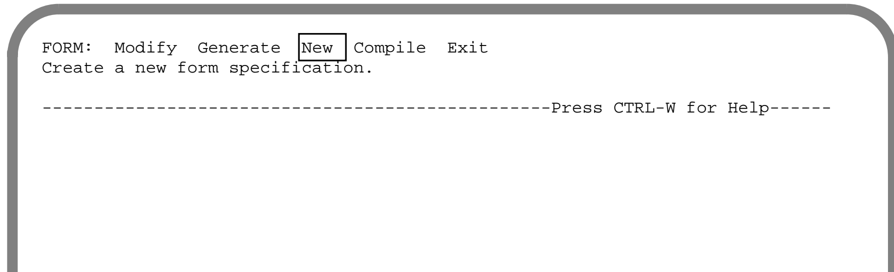
```
FORM:  Modify  Generate  New  Compile  Exit
Generate and compile a default form specification.

-------------------------------------------------Press CTRL-W for Help------
```

When you select this option, **INFORMIX-4GL** prompts you to select a database, to choose a filename for the form specification, and to identify the tables that the form will access. After you provide this information, **INFORMIX-4GL**

creates and compiles a form specification file. This is equivalent to running the **-d** (default) option of the **form4gl** command, as described in the section titled "Compiling a Form Through the Operating System" on page 5-74.

The *New* Option

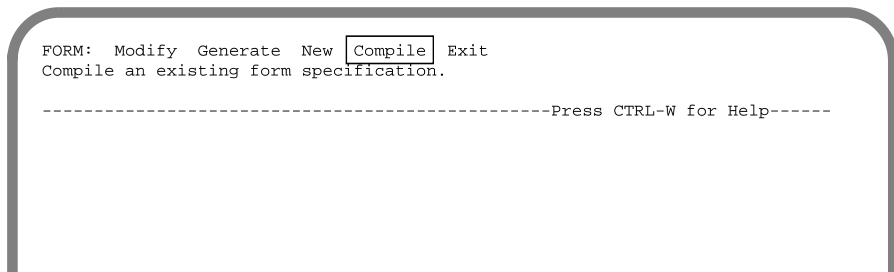
The **New** option of the FORM Design menu enables you to create a new screen form specification.



After prompting you for the name of your form specification file, **INFORMIX-4GL** places you in the editor where you can create a form specification file. When you leave the editor, **INFORMIX-4GL** transfers you to the NEW FORM menu that is like the MODIFY FORM menu. You can compile your form and correct it in the same way.

The *Compile* Option

The **Compile** option enables you to compile an existing form specification file without going through the **Modify** option.



INFORMIX-4GL prompts you for the name of the form specification file and then performs the compilation. If the compilation is not successful, **INFORMIX-4GL** displays the COMPILE FORM menu with the highlight on the **Correct** option.

The *Exit* Option

The **Exit** option clears the FORM Design menu from the screen.

```
FORM:  Modify  Generate  New  Compile  [Exit]
Returns to the INFORMIX-4GL menu.

-------------------------------------------------Press CTRL-W for Help------
```

Selecting this option restores the INFORMIX-4GL menu:

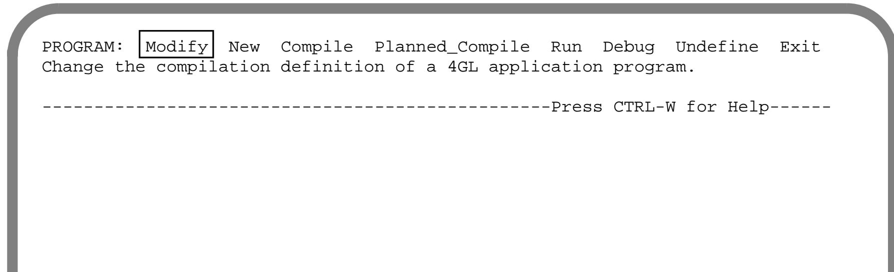
```
INFORMIX-4GL:  [Module]  Form  Program  Query-language  Exit
Create, modify or run individual 4GL program modules.

-------------------------------------------------Press CTRL-W for Help------
```

The PROGRAM Design Menu

An **INFORMIX-4GL** program can be a single source-code module that you create and compile at the MODULE Design menu. For applications of greater complexity, however, it is often easier to develop and maintain an **INFORMIX-4GL** program that includes several modules. The INFORMIX-4GL menu includes the **Program** option so that you can create multiple-module programs. When you select this option, **INFORMIX-4GL** searches your DBPATH directories (see Appendix D) for the program specification database, called **syspgm4gl**. This database describes the runner options and the modules of your program.

If **INFORMIX-4GL** cannot find this database, you are asked if you want one created. If you enter `y` in response, **INFORMIX-4GL** creates the **syspgm4gl** database, grants CONNECT privilege to PUBLIC, and displays the PROGRAM Design menu. As Database Administrator of **syspgm4gl**, you can later restrict the access of other users.

If **syspgm4gl** already exists, the PROGRAM Design menu appears.



You can use this menu to create or modify a multi-module **4GL** program specification, to compile a program, or to execute or analyze a program.

The PROGRAM Design menu supports the following eight options:

Modify	Change an existing program specification.
New	Create a new program specification.
Compile	Compile an existing program.
Planned_Compile	Display the steps to compile an existing program.
Run	Execute an existing program.
Debug	Invoke the **INFORMIX-4GL Interactive Debugger**.
Undefine	Delete an existing program specification.
Exit	Return to the INFORMIX-4GL menu.

You must first use the MODULE Design menu and FORM Design menu to enter and edit the **INFORMIX-4GL** statements within the component source-code modules of a **4GL** program. Then you can use the PROGRAM Design menu to identify which modules are part of the same application program, and to combine all the **4GL** modules in an executable program.

The *Modify* Option

The **Modify** option enables you to modify the specification of an existing **4GL** program. (This option is not valid unless at least one program has already been specified. If none has, you can create a program specification by selecting the **New** option from the same menu.) **INFORMIX-4GL** prompts

you for the name of the program specification you want to modify. It then displays a screen and menu that you can use to update the information in the program specification database, as shown in Figure 1-3:

```
MODIFY PROGRAM: 4GL Globals Other Program_Runner Rename Exit
Edit the 4GL sources list.

-----------------------------------------------Press CTRL-W for Help------
 Program  [myprog  ]
 Runner   [fglgo   ]   Runner Path   [           ]
 Debugger [fgldb   ]   Debugger Path [           ]

 4gl Source     4gl Source Path
 [main      ]   [/u/john/appl/4GL                        ]
 [funct     ]   [/u/john/appl/4GL                        ]
 [rept      ]   [/u/john/appl/4GL                        ]
 [          ]   [                                        ]
 [          ]   [                                        ]

 Global Source  Global Source Path
 [          ]   [                                       ]
 [          ]   [                                       ]

 Other .4go     Other .4go Path
 [obj       ]   [                                       ]
 [          ]   [                                       ]
```

Figure 1-3 ***Example of a Program Specification Entry***

The name of the program appears in the Program field. In Figure 1-3 this name is **myprog**. You can change the name by selecting the **Rename** option. The program name, with extension **.4gi**, is assigned to the program produced by compiling and combining all the source files. (Compiling and combining is done by the **Compile** option, as described in "The Compile Option" on page 1-49, or by the **Program_Compile** option of the MODULE Design menu.) In this case, the runable program would have the name **myprog.4gi**.

The **4GL** option enables you to update the entries for 4gl Source and 4gl Source Path. The five rows of fields under these labels form a screen array. If you select the **4GL** option, **INFORMIX-4GL** executes an INPUT ARRAY statement so you can move through the array and scroll for up to a maximum of 100 entries.

The INPUT ARRAY statement description in Chapter 3 explains how to use function keys to scroll, delete rows, and insert new rows. (You cannot redefine function keys, however, as you can with an **INFORMIX-4GL** program.)

In the example shown in Figure 1-3, the **4GL** source program has been broken into three modules: a module containing the MAIN program block (**main.4gl**), a module containing functions (**funct.4gl**), and a module contain-

ing REPORT statements (**rept.4gl**). These modules are all located in the directory **/u/john/appl/4GL**. If a module contains only global variables, you can list it here or in the Global Source array.

The **Globals** option enables you to update the Global Source array. If you use the Global Source array to store a globals module, any modification of the globals module file causes all **4GL** modules to be recompiled when you select the **Compile** option.

The **Other** option enables you to update the entries for the Other .4go and Other .4go Path fields. This is where you specify the name and location of other **4GL** object files (**.4go** files) to include in your program. Do not specify the filename extensions. You can list up to 100 files in this array.

The **Program_Runner** option enables you to specify the name and location of the p-code runner to execute your program. You can run **INFORMIX-4GL** programs with **fglgo** (the default) or with a *customized p-code runner.* A customized p-code runner is an executable program that you create to run **4GL** programs that call C functions (see "RDS Programs that Call C Functions" on page 1-64). If you do not modify the Runner field, your program is executed by **fglgo** when you select the **Run** option from the PROGRAM Design menu.

The MODIFY PROGRAM screen form contains two additional fields labeled Debugger and Debugger Path. If you have the **INFORMIX-4GL Interactive Debugger**, you can also use the **Program_Runner** option to enter the name of a customized **Debugger**. See "RDS Programs that Call C Functions" on page 1-64 for information about the use of a customized **Debugger**. For the procedures to create a customized **Debugger**, refer to Appendix C of the *Guide to the INFORMIX-4GL Interactive Debugger*, which includes an example.

The **Exit** option of the MODIFY PROGRAM menu returns you to the PROGRAM Design menu.

The *New* Option

The **New** option of the PROGRAM Design menu enables you to create a new specification of the program modules and libraries that make up the desired application program.

```
PROGRAM:  Modify  [New]  Compile  Planned_Compile  Run  Debug  Undefine  Exit
Add the compilation definition of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

The **New** option is identical to the **Modify** option, except that you must first supply a name for your program. **INFORMIX-4GL** then displays a blank form with a NEW PROGRAM menu that has the same options as the MODIFY PROGRAM menu.

The *Compile* Option

The **Compile** option compiles and combines the modules listed in the program specification database, taking into account the time when files were last updated. **INFORMIX-4GL** compiles only those files that have been modified since they were last compiled, except in the case where you have modified a module listed in the Global Source array. If you have modified a module that is listed in the Global Source array, all files are recompiled.

```
PROGRAM:  Modify  New  [Compile]  Planned_Compile  Run  Debug  Undefine  Exit
Compile a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

The **Compile** option produces a runable p-code file with a **.4gi** extension. **INFORMIX-4GL** lists each step of the compilation as it occurs.

The *Planned_Compile* Option

Taking into account the time of last change for the various files in the dependency relationships, the **Planned_Compile** option prompts for a program name and displays a summary of the steps that will be executed if you select **Compile**. No compilation actually takes place.

```
PROGRAM:  Modify  New  Compile  [Planned_Compile]  Run  Debug  Undefine  Exit
Show the planned compile actions of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
Compiling INFORMIX-4GL sources:
                /u/john/appl/4GL/main.4gl
                /u/john/appl/4GL/funct.4gl
                /u/john/appl/4GL/rept.4gl
Linking other objects:
                /u/john/appl/Com/obj.4go
```

If you have made changes in all the components of the program listed in Figure 1-3 since the last time they were compiled, **INFORMIX-4GL** displays the previous screen.

The *Run* Option

Select the **Run** option to execute a compiled program.

```
PROGRAM:  Modify  New  Compile  Planned_Compile [Run] Debug  Undefine Exit
Execute a 4GL application program

-------------------------------------------------Press CTRL-W for Help------
```

The screen lists any compiled programs (files with the extension **.4gi**) and highlights the current program, if one has been specified. This option resembles the **Run** option of the MODULE Design menu.

Although **.4go** files are not displayed, you can also enter the name and extension of a **.4go** file. Whatever compiled program you select is executed by **fglgo**, or by the runner that you specified in the Runner field of the Program Specification screen. This screen was illustrated earlier, in the description of the MODIFY PROGRAM menu.

The *Debug* Option

The **Debug** option works like the **Run** option but enables you to examine a **4GL** program with the **INFORMIX-4GL Interactive Debugger**. This option is not implemented unless you have purchased the **Debugger**.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run [Debug]  Undefine  Exit
Drop the compilation definition of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

The *Undefine* Option

The **Undefine** option of the PROGRAM Design menu prompts you for a program name and removes the compilation definition of that program from the **syspgm4gl** database. This action removes the definition only. Your program and **4GL** modules are *not* removed.

```
PROGRAM:  Modify  New  Compile  Planned_Compile  Run  Debug [Undefine]  Exit
Drop the compilation definition of a 4GL application program.

-------------------------------------------------Press CTRL-W for Help------
```

The *Exit* Option

The **Exit** option clears the PROGRAM Design menu from the screen and restores the INFORMIX-4GL menu.

The QUERY LANGUAGE Menu

The <vk>SQL interactive interface is identical to the interactive <vk>SQL interface of **INFORMIX-SQL**. You can use this option only if you have separately purchased and installed **INFORMIX-SQL** on your system.

The **Query-language** option is placed at the top-level menu so you can test <vk>SQL statements without leaving the **INFORMIX-4GL** Programmer's Environment. You can also use this option to create, execute, and save <vk>SQL scripts.

Creating Programs in the Programmer's Environment

Enter the following command at the system prompt to invoke the Programmer's Environment:

```
r4gl
```

After a sign-on message, the INFORMIX-4GL menu appears.

Creating a **4GL** application with the **INFORMIX-4GL Rapid Development System** requires the following steps:

1. Creating a new source module or revising an existing source module.
2. Compiling the source module.
3. Linking the program modules.
4. Executing the compiled program.

This process is described below.

Creating a New Source Module

This section outlines the procedure for creating a new module. If your source module already exists but needs to be modified, skip ahead to the next section, "Revising an Existing Module."

1. Select the **Module** option of the INFORMIX-4GL menu by pressing `m` or by pressing RETURN.

   The MODULE Design menu is displayed.

2. If you are creating a new **.4gl** source module, press `n` to select the **New** option of the MODULE Design menu.
3. Enter a name for the new module.

   The name must begin with a letter, and can include letters, numbers, and underscores. The name must be unique among the files in the same directory, and among the other program modules, if it will be part of a multi-module program. **INFORMIX-4GL** attaches extension **.4gl** to this identifier, as the filename of your new source module.

4. Press RETURN.

Revising an Existing Module

If you are revising an existing **4GL** source file, follow these steps:

1. Select the **Modify** option of the MODULE Design menu.

   The screen lists the names of all the **.4gl** source modules in the current directory and prompts you to select a source file to edit.

2. Use the Arrow keys to highlight the name of a source module and press RETURN, or enter a filename (with no extension).

   If you specified an editor with the DBEDIT environment variable, an editing session begins automatically. Otherwise, the screen prompts you to specify a text editor.

   Specify the name of a text editor, or press RETURN for **vi**, the default editor. Now you can begin an editing session by entering **4GL** statements. (The chapters that follow describe **INFORMIX-4GL** statements and programs.)

3. When you have finished entering or editing your **4GL** code, use an appropriate editor command to save your source file and end the text editing session.

Compiling a Source Module

The **.4gl** source file module that you create or modify is an ASCII file that must be compiled before it can be executed.

1. Select the **Compile** option from the MODULE Design menu.

2. Select the type of module you are compiling, either **Object** or **Runable**.

   If the module is a complete **4GL** program that requires no other modules, select **Runable**. This option creates a compiled p-code version of your program module, with the same filename, but with extension **.4gi**.

   If the module is one module of a multi-module **4GL** program, select **Object**. This creates a compiled p-code version of your program module, with the same filename, but with extension **.4go**. See also "Combining Program Modules" on page 1-54.

3. If the compiler detects errors, no compiled file is created, and you are prompted to fix the problem.

   Select **Correct** to resume the previous text editing session, with the same **4GL** source code, but with error messages in the file. Edit the file to correct the error, and select **Compile** again. If an error message appears, repeat this process until the module compiles without error.

4. After the module compiles successfully, select **Save-and-exit** from the menu to save the compiled program.

   The MODULE Design menu appears again on your screen.

5. If your program requires screen forms, select **Form** from the INFORMIX-4GL menu to display the FORM Design menu. For information about designing and creating screen forms, see Chapter 5.

6. If your program displays help messages, you must create and compile a help file.

   Use the **mkmessage** utility to compile the file. For more information about this utility, see Appendix B.

Combining Program Modules

If your new or modified module is part of a multi-module **4GL** program, you must combine all of the modules into a single program file before you can run the program. If the module that you compiled is the only module in your program, you are now ready to run your program (see "Executing a Compiled RDS Program" on page 1-56).

1. Select the **Program** option from the INFORMIX-4GL menu.

   The PROGRAM Design menu appears.

2. If you are creating a new multi-module **4GL** program, select the **New** option. If you are modifying an existing one, select **Modify**.

   In either case, the screen prompts you for the name of a program.

3. Enter the name (without a file extension) of the program that you are modifying, or the name to be assigned to a new program.

   Names must begin with a letter, and can include letters, underscores (_), and numbers. After you enter a valid name, the PROGRAM screen appears, with your program name in the first field.

   If you selected **Modify**, the names and pathnames of the source-code modules are also displayed. The PROGRAM screen appears below the

MODIFY PROGRAM menu, rather than below the NEW PROGRAM menu. (Both menus list the same options.)

```
NEW PROGRAM:  4GL  Globals  Other  Program_Runner  Rename  Exit
Edit the 4GL sources list.

---------------------------------------------- Press CTRL-W for Help -------
 Program [          ]
 Runner  [fglgo     ] Runner Path   [                                      ]
 Debugger[fgldb     ] Debugger Path [                                      ]

 4gl Source    4gl Source Path
 [          ]  [                                      ]
 [          ]  [                                      ]
 [          ]  [                                      ]
 [          ]  [                                      ]
 [          ]  [                                      ]

 Global Source Global Source Path
 [          ]  [                                      ]
 [          ]  [                                      ]

 Other .4go    Other .4go Path
 [          ]  [                                      ]
 [          ]  [                                      ]
```

4. Identify the files that comprise your program:
   - To specify new **4GL** modules or edit the list of **4GL** modules, select the **4GL** option.

     You can enter or edit the name of a module under the heading `4GL Source`; the **.4gl** file extension is optional. Repeat this step for every module. If the module is not in the current directory or in a directory specified by the DBPATH environment variable, enter the pathname to the directory where the module resides.

     The name of the runner (and of the **Debugger**, if you have the **INFORMIX-4GL Interactive Debugger**) are usually as illustrated in the PROGRAM screen, unless your **4GL** program calls C functions. For information on calling C functions, see "RDS Programs that Call C Functions" on page 1-64.
   - To enter or edit the name or pathname of a **Globals** module, select the **Globals** option and provide the corresponding information.
   - To enter or edit the file or pathname of any **.4go** modules that you have already compiled, select the **Other** option.
5. After you correctly list all of the modules of your **4GL** program, select the **Exit** option to return to the PROGRAM Design menu.

6. Select the **Compile** option of the PROGRAM Design menu.

   This option produces a file that combines all of your **.4gl** source files into an executable program. Its filename is the program name that you specified, with extension **.4gi**. The screen lists the names of your **.4gl** source modules and displays the PROGRAM Design menu with the **Run** option highlighted.

Executing a Compiled RDS Program

After compiling your program modules, you can execute your program. To do so, select the **Run** option from the MODULE Design menu. This option executes the compiled **4GL** program.

Menus, screen forms, windows, or other screen output are displayed, according to your program logic and your keyboard interaction with the program.

Invoking the Debugger

If you are developing or modifying an **INFORMIX-4GL** program, you have much greater control over program execution by first invoking the **INFORMIX-4GL Interactive Debugger**. If you have purchased the **Debugger**, you can invoke it from the MODULE Design menu or PROGRAM Design menu of the Programmer's Environment by selecting the **Debug** option.

For information on using the **Debugger** as a programmer's productivity tool, see the *Guide to the INFORMIX-4GL Interactive Debugger*.

Creating Programs at the Command Line

You can also create **.4gl** source files and compiled **.4go** and **.4gi** p-code files at the operating system prompt. Figure 1-4 shows the process of creating, compiling, and running or debugging a single-module program from the command line.

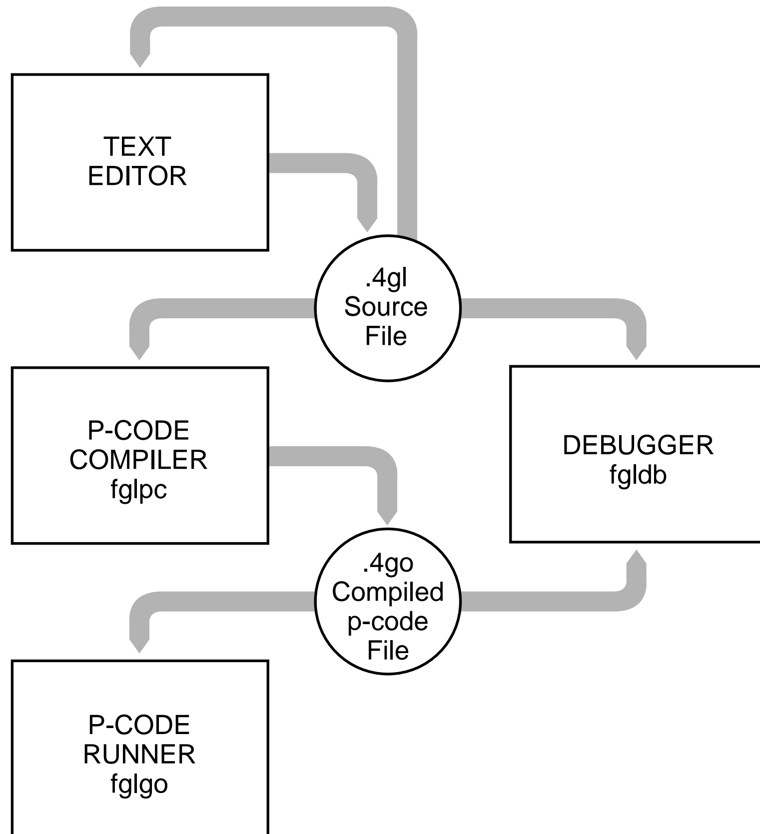


Figure 1-4 ***Creating and Running a Single-Module Program***

In Figure 1-4 the rectangles represent processes controlled by specific commands, and the circles represent files. Arrows indicate whether a file serves as input or output for a process.

This diagram is simplified and ignores the similar processes by which forms, help messages, and any other components of **INFORMIX-4GL** applications are compiled and executed.

- The cycle begins in the upper left corner with a text editor, such as **vi**, to produce a **4GL** source module.
- The program module can then be compiled, using the **fglpc** p-code compiler. (If error messages are produced by the compiler, find them

in the **.err** file, and edit the **.4gl** file to correct the errors. Then recompile the corrected **.4gl** file.)

- The following command line invokes the p-code runner:

```
fglgo filename
```

where *filename* specifies a compiled **4GL** file to be executed.

Executing a program that is undergoing development or modification sometimes reveals the existence of run-time errors. If you have licensed the **INFORMIX-4GL Interactive Debugger**, you can invoke it to analyze and identify run-time errors in your program by entering the command:

```
fgldb filename
```

where *filename* specifies your compiled **4GL** file. You can then recompile and retest the program. When it is ready for use by others, they can use the **fglgo** runner to execute the compiled program.

A correspondence between commands and menu options of the **RDS** Programmer's Environment is summarized by the following list:

Command	Invokes	Menu Option
vi	UNIX System Editor	Module New/Modify
fglpc	4GL P-Code Compiler	Compile
fglgo	4GL P-Code Runner	Run
fgldb	4GL Interactive Debugger	Debug

Subsequent sections of this chapter describe how to use the **INFORMIX-4GL Rapid Development System** to compile and execute **4GL** programs that call C functions. (These special **Rapid Development System** procedures require a C language compiler and linker, which are unnecessary for **4GL** applications that do not call programmer-defined C functions.)

Creating or Modifying a 4GL Source File

Use your system editor or another text-editing program to create a **.4gl** source file, or to modify an existing file. For information on the statements you can include in a **4GL** program, see Chapter 3.

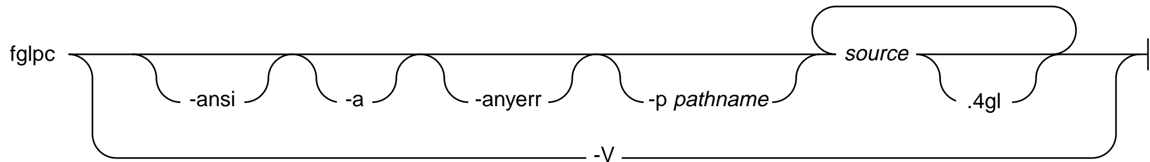
Compiling an RDS Source File

You cannot execute a **4GL** program until you have compiled each source module into a **.4go** file. Do this at the system prompt by entering the **fglpc** command, which compiles your **4GL** source code, and generates a file

containing tables of information and blocks of p-code. You can then run this compiled code by using the **INFORMIX-4GL** p-code runner (or the **INFORMIX-4GL Interactive Debugger**, if you have the **Debugger**).

The **INFORMIX-4GL** source-code module to be compiled should have the file extension **.4gl**.

fglpc Command



pathname is the pathname of the directory to contain object and error files.

source is the name of an **INFORMIX-4GL** source-code module. The **.4gl** extension is optional.

Usage

The **fglpc** command reads *source*.**4gl** files and creates a compiled version of each, with the filename *source*.**4go**. You can specify any number of source files, in any order, with or without their **.4gl** filename extensions.

To instruct the compiler to check all <vk>SQL statements for compliance with ANSI standards, use the -**ansi** option. If you specify the -**ansi** option, it must appear first in your list of **fglpc** command arguments. The -**ansi** option asks for compile-time warning messages if your source code includes Informix extensions to the ANSI standard for <vk>SQL.

If an error or warning occurs during compilation, **INFORMIX-4GL** creates a file called *source*.**err**. Look in *source*.**err** to find where the error or warning occurred in your code.

If you specify the -**anyerr** option, 4GL sets the **status** variable after evaluating expressions. The -**anyerr** option overrides any WHENEVER ERROR statements in your program.

You can use the -**p** *pathname* option to specify a non-default directory for the object (**.4go**) and error (**.err**) files. Otherwise, any files produced by **fglpc** are stored in your current working directory.

To have your compiled program check array bounds at run time, specify the -**a** option. The -**a** option requires additional processing, so you may want to use this option only for debugging during development.

To display the version number of the software, specify the -**V** option. The version number of your <vk>SQL and p-code compiler software appears on the screen. Any other command options are ignored. After displaying this information, the program terminates without compiling.

Examples

The following command compiles a **4GL** source file **single.4gl**, and creates a file called **single.4go** in the current directory:

```
fglpc single.4gl
```

The next command line compiles two **4GL** source files:

```
fglpc -p /u/ken fileone filetwo
```

This command generates two compiled files, **fileone.4go** and **filetwo.4go**, and stores them in subdirectory **/u/ken**. Any compiler error messages are saved in files **fileone.err** or **filetwo.err** in the same directory.

Concatenating Multi-Module Programs

If a program has several modules, the compiled modules must all be concatenated into a single file, as represented in Figure 1-5:

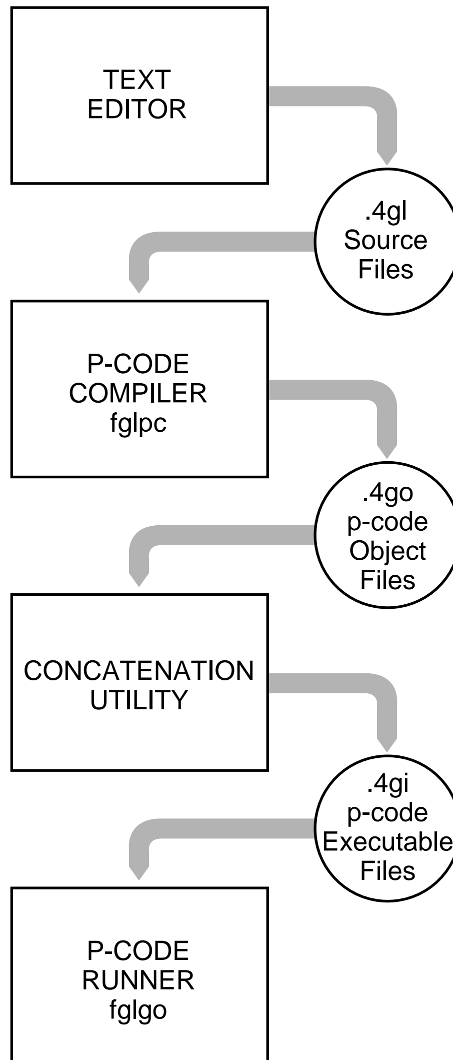


Figure 1-5 ***Creating and Running a Multi-Module Program***

The UNIX **cat** command combines the listed files into the file specified after the redirect symbol (>). For example, the following command combines a list of **.4go** files into a new file called **new.4gi**:

```
cat file1.4go file2.4go ... fileN.4go > new.4gi
```

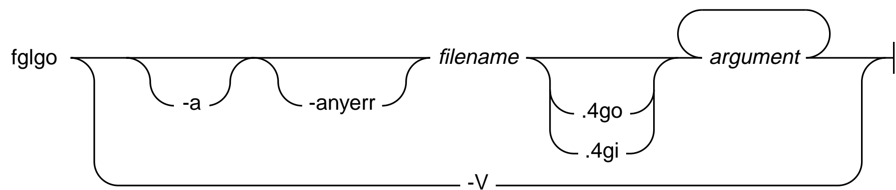
Note: *The new filename of the combined file must have either a **.4go** or a **.4gi** extension. Throughout this manual, extension **.4gi** designates runable files that have been compiled (and concatenated, if several source modules comprise the program). You may wish to follow this convention in naming files, since only **.4gi** files are displayed from within the Programmer's Environment. This convention is also a convenient way to distinguish complete program files from object files that are individual modules of a multi-module program.*

If your **4GL** *program calls C functions or* **INFORMIX-ESQL/C** *functions, you must follow procedures described in* *"RDS Programs that Call C Functions" on page 1-64* *before you can run your application.*

Running RDS Programs

To execute a compiled **4GL** program from the command line, you can invoke the p-code runner, **fglgo**.

fglgo Command



filename is the name of a compiled **4GL** file. The *filename* must have a **.4go** or **.4gi** extension. You do not need to enter this extension on the command line.

argument are any arguments required by your **4GL** program.

Usage

If you do not specify a filename extension in the command line, **fglgo** looks first for the *filename* with a **.4gi** extension, and then for the *filename* with a **.4go** extension.

To have your compiled program check array bounds at run time, specify the -**a** option. The -**a** option requires additional processing, so you may want to use this option only for debugging during development.

If you specify the -**anyerr** option, 4GL sets the **status** variable after evaluating expressions. The -**anyerr** option overrides any WHENEVER ERROR statements in your program.

To display the version number of the software, specify the **-V** option. The version number of your <vk>SQL and p-code software appears on the screen. Any other command options are ignored. After displaying this information, the program terminates without invoking the p-code runner.

Note: *To run a* **4GL** *program that calls programmer-defined C functions, you cannot use* ***fglgo****. You must instead use a customized p-code runner. The section* *"RDS Programs that Call C Functions" on page 1-64* *describes how to create a customized runner.*

Examples

To run a compiled program named **myprog.4go**, enter the following command line at the operating system prompt:

```
fglgo myprog
```

or:

```
fglgo myprog.4go
```

Running Multi-Module Programs

To run a program with multiple modules, you must compile each module and then combine them by an operating system concatenation utility, as described in an earlier section. For example, if **mod1.4go**, **mod2.4go**, and **mod3.4go** are compiled **INFORMIX-4GL** modules that you want to run as one program, you must first combine them as in the following example:

```
cat mod1.4go mod2.4go mod3.4go > mods.4gi
```

You can then run the **mods.4gi** program by using the command lines:

```
fglgo mods
```

or:

```
fglgo mods.4gi
```

Running Programs with the Interactive Debugger

You can also run compiled **4GL** programs with the **INFORMIX-4GL Interactive Debugger**. This **4GL** source-code debugger is a p-code runner with a rich command set for analyzing **4GL** programs. You can use the **Debugger** to locate logical and run-time errors in your **4GL** programs and to become more familiar with **4GL** programs. The **Debugger** must be purchased separately from **INFORMIX-4GL**.

If you have the **Debugger**, you can invoke it at the system prompt by a command line of the form:

```
fgldb filename
```

where *filename* is any runable **4GL** file that you produced by an **fglpc** command.

For the complete syntax of the **fgldb** command, see Chapter 8 of the *Guide to the INFORMIX-4GL Interactive Debugger*.

RDS Programs that Call C Functions

If your **INFORMIX-4GL Rapid Development System** program calls programmer-defined C functions, you must create a customized runner to execute the program. You can do this by following two steps:

1. Edit a structure definition file to contain information about your C functions.

   This file is named **fgiusr.c** and is supplied with **INFORMIX-4GL**.
2. Compile and link the **fgiusr.c** file with the files that contain your C functions.

   To do this, use the **cfglgo** command.

You can then use the runner produced by the `cfglgo` command to run the **4GL** program that calls your C functions. Both the **fgiusr.c** file and the `cfglgo` command are described in the pages that follow.

For an example of how to call C functions from a **4GL** program, see Example 13, "Calling a C Function," in *INFORMIX-4GL by Example*.

Note: *To create a customized runner, you must have a C compiler installed on your system. If the only functions that your* **INFORMIX-4GL Rapid Development System** *program calls are* **INFORMIX-4GL** *or* **INFORMIX-ESQL/C** *library functions, or functions written in the* **INFORMIX-4GL** *language, you do not need a C compiler and you do not need to follow the procedures described in this section.*

Editing the *fgiusr.c* File

With your **INFORMIX-4GL** software, you receive a file named **fgiusr.c**. This file is located in the **/etc** subdirectory of the directory in which you install **INFORMIX-4GL** (that is, in **$INFORMIXDIR/etc**). The following listing shows the **fgiusr.c** file in its unedited form:

```
/***************************************************************
*                                                             *
*            INFORMIX SOFTWARE, INC.                          *
*                                                             *
* Title: fgiusr.c                                             *
* Sccsid: @(#)fgiusr.c   4.2   8/26/87  10:48:37              *
* Description:                                                *
*       definition of user C functions                        *
*                                                             *
***************************************************************
*/

/***********************************************************
* This table is for user-defined C functions.
*
* Each initializer has the form:
*
*   "name", name, nargs
*
* Variable # of arguments:
*
*   set nargs to -(maximum # args)
*
* Be sure to declare name before the table and to leave the
* line of 0's at the end of the table.
*
* Example:
*
*   You want to call your C function named "mycfunc" and it expects
*   2 arguments.  You must declare it:
*
*       int mycfunc();
*
*   and then insert an initializer for it in the table:
*
*       "mycfunc", mycfunc, 2
*************************************************************
*/

#include "fgicfunc.h"

cfunc_t usrcfuncs[] =
    {
    0, 0, 0
    };
```

The **fgiusr.c** file is a C language file that you can edit to declare any number of programmer-defined C functions.

To edit **fgiusr.c**, you can copy the file to any directory. (Unless this is your working directory at compile time, you must specify the full pathname of the edited **fgiusr.c** file when you compile.) Edit **fgiusr.c** to specify the following:

- A declaration for each function:

  `int` *function-name*`();`

- Three initializers for each function:

  " *function-name* ", *function-name*, [-] *integer*,

In the declaration of the function, the parenthesis symbols `( )` must follow the *function-name*.

The first initializer is the function name between double quotation marks and is a *character pointer*.

The second initializer is the function name without quotes and is a *function pointer*.

The third initializer is an *integer* representing the number of arguments expected by the function. If the number of arguments expected by the function can vary, make the third argument the maximum number of arguments, prefixed with a minus (-) sign.

You must use commas (,) to separate the three initializers. Insert a set of initializers for each C function that you declare. A line of three zeroes indicates the end of the structure.

Here is an example of an edited **fgiusr.c** file:

```
#include "fgicfunc.h"

int function-name();

cfunc_t usrcfuncs[] =
{
"function-name",function-name,1, 0,0,0
}
```

Here the **4GL** program will be able to call a single C function called *function-name* that has one (1) argument.

If you have several **4GL** programs that call C functions, you can use **fgiusr.c** in either of two ways:

- You can create one customized p-code runner.

  In this case, you can edit **fgiusr.c** to specify all the C functions called from all your **4GL** programs. After you create one comprehensive runner, you can use it to execute all your **4GL** applications.

- You can create several application-specific runners.

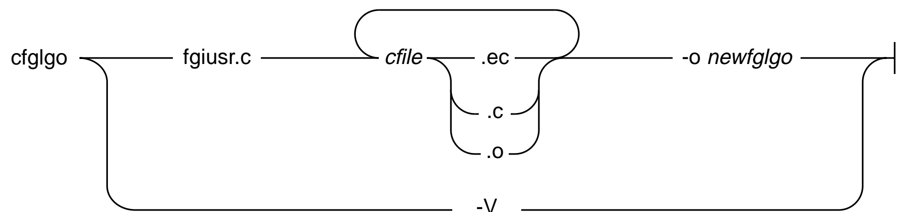
  In this case, you can either make a copy of the **fgiusr.c** file (with a new name) for each customized runner, or you can re-edit **fgiusr.c** to contain information on the C functions for a specific application before you compile and link. If you create several runners, you must know which customized runner to use with each **4GL** application.

In some situations the first method is more convenient, since users do not have to keep track of which runner supports each **4GL** application.

Creating a Customized Runner

You can use the **cfglgo** command to create a customized runner. You can use **cfglgo** to compile C modules and **INFORMIX-ESQL/C** modules that contain functions declared in an edited **fgiusr.c** file. The customized runner can also run **4GL** programs that do not call C functions.

cfglgo Command



cfile is the name of a source file containing **INFORMIX-ESQL/C** or C functions to be compiled and linked with the new runner; or the name of an object file previously compiled from a **.c** or **.ec** file. You can specify any number of uncompiled or compiled C or **INFORMIX-ESQL/C** files in a **cfglgo** command line.

newfglgo specifies the name of the customized runner.

Usage

You must have the **INFORMIX-ESQL/C** product to compile **INFORMIX-ESQL/C** files with **cfglgo**.

The **cfglgo** command compiles and links the edited **fgiusr.c** file with your C program files into an executable program that can run your **4GL** application. **fgiusr.c** is the name of the file that you edited to declare C and/or **INFORMIX-ESQL/C** functions. If the **fgiusr.c** file to be linked is not in the current directory, you must specify a full pathname. You can also rename the **fgiusr.c** file. If you do so, specify its new filename in place of **fgiusr.c**.

If you do not specify the **-o** *newfglgo* option, the new runner is given the default name **a.out**.

To display the version number of the software, specify the **-V** option. The version number of your <vk>SQL and p-code software appears on the screen. Any other command options are ignored. After displaying this information, the program terminates without creating a customized p-code runner.

Examples

The following example **4GL** program calls the C function **prdate( )**:

---

prog.4gl:

```
main
. . .
call prdate()
. . .
end main
```

The function **prdate( )** is defined in file **cfunc.c**, as shown here:

cfunc.c:

```
#include <stdio.h>
#include <time.h>

prdate()
{
/* This program timestamps file FileX */

  long cur_date;
  extern int errno;
  FILE *fptr;

  time(&cur_date);
  fptr = fopen("time_file","a");
  fprintf(fptr,"FileX was accessed %s", ctime(&cur_date));
  fclose(fptr);
}
```

The C function is declared and initialized in the following **fgiusr.c** file:

fgiusr.c:

```
1 #include "fgicfunc.h"
2
3 int prdate();
4 cfunc_t usrcfuncs[] =
5     {
6   "prdate", prdate, 0,
7    0, 0, 0
8    };
```

An explanation of this example of an **fgiusr.c** file follows:

line 1: The file **fgicfunc.h** is always included. This line already exists in the unedited **fgiusr.c** file.

line 3: This is the declaration of the function **prdate( )**. You must add this line to the file.

line 4: This line already exists in the unedited file. It declares the structure array *usrcfuncs*.

line 6: This line contains the initializers for function **prdate( )**. Since it expects no arguments, the third value is zero.

line 7: The line of three zeros indicates that no more functions are to be included.

In this example, you can use the following commands to compile the **4GL** program, to compile the new runner, and to run the program:

To compile the example 4GL program:

```
fglpc prog.4gl
```

To compile the new runner:

```
cfglgo fgiusr.c cfunc.c -o newfglgo
```

To run the 4GL program:

```
newfglgo prog.4go
```

Running Programs that Call C Functions

After you create a customized runner, you can use it to execute any **4GL** program whose C functions you correctly specified in the edited **fgiusr.c** file. The syntax of a customized runner (apart from its name) is the same as the syntax of **fglgo**, which is described in "Running RDS Programs" on page 1-62.

You can also create a customized **Debugger** to run a **4GL** program that calls C functions. See Appendix C of the *Guide to the INFORMIX-4GL Interactive Debugger* for details and an example of how to create a customized **Debugger**.

Note: *You cannot create a customized runner or a customized* **Debugger** *from within the Programmer's Environment. You must work from the system prompt and follow the procedures described in the section titled* *"Creating Programs at the Command Line" on page 1-56* *if you are developing a* **4GL** *program that calls user-defined C functions. Then you can return to the Programmer's Environment and use the* ***Program_Runner*** *option of the MODIFY PROGRAM menu or NEW PROGRAM menu to specify the name of a customized runner or* **Debugger**.

Program Filename Extensions

Source, runable, error, and backup files generated by **INFORMIX-4GL** are stored in the current directory and are labeled with a filename extension. The following list shows the file extensions for the source, runable, and error files. These files are produced during the normal course of using the **INFORMIX-4GL Rapid Development System**.

File	Description
file.4gl	**4GL** source file.
file.4go	**4GL** file that has been compiled to p-code.
file.4gi	**4GL** file that has been compiled to p-code.
file.err	**4GL** source error file, created when an attempt to compile a module fails or produces a warning. The file contains the **4GL** source code plus compiler syntax warnings or error messages.
file.erc	**4GL** object error file, created when an attempt to compile or to link a non-**INFORMIX-4GL** source-code or object module fails. The file contains **4GL** source code and annotated compiler errors.
form.per	**FORM4GL** source file.
form.frm	**FORM4GL** object file.
form.err	**FORM4GL** source error file.

The last three files do not exist unless you create or modify a screen form specification file, as described in Chapter 5.

The following list identifies backup files that are produced when you use **INFORMIX-4GL** from the Programmer's Environment.

File	Description
file.4bl	**4GL** source backup file, created during the modification and compilation of a **.4gl** program module.
file.4bo	Object backup file, created during the compilation of a **.4go** program module.
file.4be	Object backup file, created during the compilation of a **.4gi** program module.
file.pbr	**FORM4GL** source backup file.
file.fbm	**FORM4GL** object backup file.

Under normal conditions, **INFORMIX-4GL** creates the backup files and intermediate files as necessary, and deletes them upon a successful compilation. If you interrupt a compilation, you may find one or more of the files in your current directory.

If you compile with a **fglpc** command line that includes the **p** *pathname* option, **INFORMIX-4GL** creates the **.4gi, .4go, .err,** and corresponding backup files in the directory specified by *pathname*, rather than in your current directory.

During the compilation process, **INFORMIX-4GL** stores a backup copy of the **file.4gl** source file in **file.4bl**. The time stamp is modified on the (original) **file.4gl** source file, but not on the backup **file.4bl** file. In the event of a system crash, you may need to replace the modified **file.4gl** file with the backup copy contained in the **file.4bl** file.

The Programmer's Environment does not allow you to begin modifying a **.4gl** or **.per** source file if the corresponding backup file already exists in the same directory. After an editing session terminates abnormally, for example, you must delete or rename any backup file before you can resume editing your **4GL** module or form from the Programmer's Environment.

Chapter

2

The INFORMIX-4GL Language

Overview of 4GL 3

Language Features 3
- Lettercase Insensitivity 3
- 4GL Statements 4
- Comments 5
  - Comment Indicators 6
  - Restrictions on Comments 6
- Source Code Modules and Program Blocks 7
- Statement Blocks 8
- Statement Segments 9
- 4GL Identifiers 9
  - Naming Rules for 4GL Identifiers 10
  - Scope of Reference of 4GL Identifiers 11
  - Scope and Visibility of SQL Identifiers 12
  - Visibility of Identical Identifiers 13

Interacting with Users 15
- Ring Menus 15
  - Selecting Menu Options 16
  - Ambiguous Keyboard Selections 16
  - Hidden Options and Invisible Options 16
  - Disabled Menus 17
  - Reserved Lines for Menus 17
- Screen Forms 17
  - Visual Cursors 18
  - Field Attributes 18
  - Reserved Lines 19
- 4GL Windows 19
  - The Current Window 20

On-Line Help 21
The Help Key and the Message Compiler 21
The Help Window 22

Exception Handling 23
Error Handling with SQLCA 23
A Taxonomy of Run-Time Errors 26

Overview of 4GL

An INFORMIX-4GL program consists of at least one source file containing a series of English-like statements. These obey a well-defined syntax that this book describes. Here the term "4GL" is a synonym for "INFORMIX-4GL."

This chapter presents a brief overview of the language. Its theory, application, constructs, and semantics are described in detail in *INFORMIX-4GL Concepts and Use*, a companion volume to this manual.

This manual assumes that you are using **INFORMIX-OnLine Dynamic Server** as your database server. Features specific to INFORMIX-SE are noted.

Language Features

INFORMIX-4GL is an English-like C or COBOL-replacement programming language that Informix Software, Inc., introduced in 1986 as a tool for creating relational DBMS applications. Its statement set (page 3-11) includes the industry-standard SQL language for accessing a relational database.

The INFORMIX-4GL development environment provides a complete environment for writing 4GL programs.

Lettercase Insensitivity

4GL makes no distinction between uppercase and lowercase letters, except in character strings enclosed within quotation marks. Use quotation marks in your source code modules if you need to preserve the lettercase of character string literals, filenames, pathnames, the names of database entities, or arguments of C function calls.

4GL is completely free-form, like C or Pascal, and generally ignores TAB characters, LINEFEED characters, comments, and extra blank spaces or lines between statements or between statement elements. You can use these non-significant characters to make your 4GL source code easier to read.

You can mix uppercase and lowercase letters in the identifiers that you assign to 4GL entities (page 2-9), but 4GL downshifts any uppercase characters in identifiers to lowercase at compile time.

4GL Statements

INFORMIX-4GL source code modules can contain statements and comments:

- A *statement* is a logical unit of code within 4GL programs. See page 3-11 for a list of statements in the INFORMIX-4GL statement set.
- A *comment* is a specification that INFORMIX-4GL disregards. See page 2-6 for information about the 4GL comment indicators.

A compilation error usually occurs if your program (or one of its modules or blocks, as described on pages 1-11 and 1-12) includes only part of a statement, but not all the required elements. Statements of 4GL can contain identifiers, keywords, literal constants, operators, and expressions. These terms are described in subsequent sections of this chapter, and in Chapter 3.

For the purposes of this manual, 4GL supports two types of statements:

- 4GL language statements
- SQL (*Structured Query Language*) statements

This distinction among statements reflects whether they provide instructions to the database server (SQL statements) or instructions to the application (other 4GL statements). See *INFORMIX-4GL Concepts and Use* for information about the process architecture of INFORMIX-4GL applications.

Within the broad division into SQL and other 4GL statements, the 4GL statement set can be further classified into several functional categories:

Types of SQL Statements	**Types of 4GL Statements**
Data definition	Definition and declaration
Data manipulation	Program flow
Dynamic management	Compiler directives
Data access	Storage manipulation
Query optimization	Report execution
Data integrity	Screen interaction
Cursor manipulation	
Stored procedure	
OnLine/Optical	

The section "The 4GL Statement Set" on page 3-11 identifies the SQL statements and other 4GL statements that comprise these functional categories.

Some statements, called *compound* statements (page 2-8), can contain other 4GL statements. A set of nested statements within a compound statement is called a *statement block*. When necessary, 4GL uses END (with another keyword to indicate a specific *statement*) to terminate a compound statement.

Except in a few special cases (like multiple-statement prepared entities, and some specifications for screen forms and for formatting reports), 4GL requires no statement terminators; but if you want, you can use the semicolon (;) as a statement terminator. (See also the PRINT statement on page 6-42, however, which can use the semicolon to control the format of output from a report.)

Note: *Screen forms of 4GL are manipulated by form drivers (page 5-3), but are defined in form specification files (page 5-6). These ASCII files use a syntax that is distinct from the syntax of other 4GL features. See Chapter 5 for details of the syntax of 4GL form specification files. Similarly, 4GL reports resemble functions, but require some special considerations; Chapter 6 describes the syntax of 4GL reports.*

Comments

A *comment* is one or more characters or lines of text that you include in 4GL source code to assist human readers, but which INFORMIX-4GL ignores. (This meaning of *comment* in documenting your program source code is unrelated to the COMMENTS attribute of a form specification file, or to the COMMENT LINE keywords of the OPTIONS statement, both of which control on-screen text displays to assist users of the application.)

For clarity and to simplify program maintenance, it is recommended that you document your 4GL code by including comments in your **.4gl** source files. You can also use comment symbols during program development to disable statements without deleting them from your source code modules.

Comment Indicators

You can indicate comments in any of three ways:

- A comment can begin with the left-brace (**{**) and end with the right-brace (**}**) symbol. These can be on the same line or on different lines.
- The pound (#) symbol (sometimes called the "sharp symbol") also can begin a comment. The comment terminates at the end of the current line.
- You can also use a pair of hyphen symbols or minus signs (--) to begin a comment that terminates at the end of the current line. (The use of this comment indicator conforms to the ANSI standard for SQL.)

All text between the braces symbols (or from the # or -- comment indicator to the end of the same line) is ignored.

Restrictions on Comments

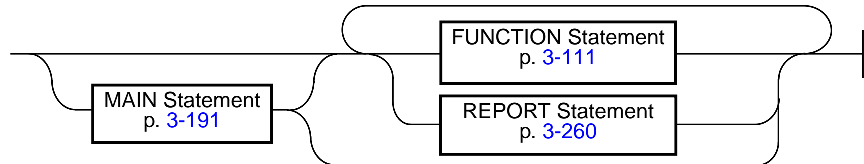
When using comments, keep the following restrictions in mind :

- Within a *quoted string*, INFORMIX-4GL interprets comment symbols as literal characters, rather than as comment indicators.
- Comments cannot appear in the SCREEN section of a form specification file, nor can the # symbol indicate comments anywhere in a form specification file.
- You cannot use the braces indicator to nest comments within comments.
- You cannot specify consecutive minus (--) signs in arithmetic expressions, because **4GL** interprets what follows as a comment. Instead use a blank space ( ) or parentheses (( )) symbols to separate consecutive arithmetic minus signs. For example:

```
LET x = y --3   # Now variable x evaluates as y
                # because 4GL ignores text after --

LET x = y -(-3) # Now variable x evaluates as (y + 3).
```

Source Code Modules and Program Blocks

When you create a 4GL program, enter statements and comments into one or more ASCII files, called *modules.* Most statements are organized into larger units, called *program blocks* (sometimes called *routines*, *sections*, or *functions*). 4GL modules can include three kinds of program blocks: MAIN, FUNCTION, and REPORT.



Each block begins with the keyword after which it is named, and ends with the corresponding END *statement* keywords (END MAIN, END FUNCTION, or END REPORT). Program blocks can support **4GL** applications in several ways:

- As part of a complete 4GL program (one that includes a MAIN block).
- As a 4GL FUNCTION or REPORT block called by a 4GL program.
- As a 4GL FUNCTION block called by a C language or **ESQL/C** program. (The INFORMIX-ESQL/C product requires a separate license.)

The following rules apply to 4GL program blocks:

- Every 4GL program must contain exactly one (1) MAIN program block. This must be the first program block of the module in which it appears.
- Except for declarations (DATABASE, DEFINE, GLOBALS), no statement can appear outside a program block.
- Variables that you declare within a program block have a scope of reference (page 2-11) that is *local* to the same program block. They cannot be referenced from other program blocks.
- The GO TO or GOTO keywords cannot reference a statement label in a different program block. (Statement labels are described in Chapter 3.)
- Program blocks cannot be nested, nor divided among different modules.
- DATABASE (page 3-58) has a compile-time effect when it appears before the first program block. Within a program block, it has a run-time effect.
- The scope of the WHENEVER statement extends from its occurrence to the end of the same module, but it must occur within a program block.

The CALL, RETURN, START REPORT, OUTPUT TO REPORT, and FINISH REPORT statements, or any expression that includes a programmer-defined function as an operand (page 3-332), can transfer control of program execution among program blocks. These statements are described in Chapter 3.

Chapter 4 describes 4GL FUNCTION blocks, and Chapter 6 describes 4GL REPORT blocks. (See Chapter 1 for details of how source code modules are compiled and linked to create applications, and naming conventions for filenames and for file extensions of 4GL modules.)

Statement Blocks

The MAIN, FUNCTION, and REPORT statements are special cases of *compound statements*, the 4GL statements that can contain other statements:

CASE	FOREACH	INPUT	PROMPT
CONSTRUCT	FUNCTION	INPUT ARRAY	REPORT
DISPLAY ARRAY	GLOBALS	MAIN	WHILE
FOR	IF	MENU	

Every compound statement of 4GL supports the END keyword to mark the end of the compound statement construct within the source code module. Most compound statements also support the EXIT *statement* keywords, to transfer control of execution to the statement that follows the END *statement* keywords, where *statement* is the name of the compound statement.

By definition, every compound statement can contain at least one *statement block*, a group of one or more consecutive SQL statements or other 4GL statements. In the syntax diagram of a compound statement, a *statement block* always includes this element:

(Some contexts permit or require the semicolon (;) to separate *statements*.) These are examples of *statement blocks* within compound 4GL statements:

- The WHEN, OTHERWISE, THEN, or ELSE blocks of CASE or IF statements.
- Statements within FOR, FOREACH, or WHILE loops.
- CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY control blocks.
- Statements following the COMMAND clauses of MENU statements.
- Statements within the ON KEY blocks of PROMPT statements.
- FORMAT section control blocks of REPORT statements.
- All the statements within a MAIN, FUNCTION, or REPORT program block.

4GL statement blocks can contain other statement blocks. This recursion can be *static*, as when a function includes a FOREACH loop that contains an IF statement. Blocks can also be recursive in a *dynamic* sense, as when a CALL statement invokes a function only if some specified condition occurs.

Although most 4GL statements can appear within statement blocks, and most compound statements can be nested, some restrictions apply. In some cases, you can circumvent these restrictions by invoking a *function* to execute a statement that cannot appear directly within a given statement.

Statement Segments

Any subset of a 4GL statement, including the entire statement, is called a *statement segment*. To simplify the description of some statements, many syntax diagrams in this book use rectangles to represent certain 4GL statement segments (for example, MAIN, FUNCTION, and REPORT on page 2-7). These are expanded into syntax diagrams on the page referenced in the rectangle, or elsewhere on the same page, if the rectangle indicates no page number.

Chapter 3 lists and describes statement segments that are elements of several 4GL statements (see page 3-289). Most of these segments are also described more tersely in the syntax articles that describe the statements in which they can occur.

4GL Identifiers

Statements and form specifications can reference some 4GL program entities by name. To create a named program entity, you must declare a 4GL *identifier*. When you create any of the following program entities, you must follow the naming rules (page 2-10) and declaration procedures of INFORMIX-4GL to declare a valid identifier:

Named Program Entity	How Name is Declared
4GL function or its formal argument	FUNCTION statement
4GL program variable	DEFINE and GLOBAL statements
4GL report or its formal argument	REPORT statement
4GL screen field	ATTRIBUTES section of form specification
4GL screen form	OPEN FORM statement
4GL screen record or screen array	INSTRUCTIONS section of form specification
4GL statement label	LABEL statement
4GL table alias	TABLES section of form specification
4GL window	OPEN WINDOW statement

This list excludes *columns, constraints, cursors, databases, indexes, prepared statements, stored procedures, synonyms, tables, triggers, views,* and other *engine objects*, because those are SQL entities, not 4GL entities. It also omits *filenames, pathnames*, and *usernames*, which must conform to operating system or network rules. Consult the documentation of your database engine or of your operating system or network for information about non-4GL identifiers.

Naming Rules for 4GL Identifiers

A 4GL identifier is a character string that is used as the name of a program entity. Every 4GL identifier must conform to the following rules:

- It must include at least one character, but no more than 50.
- It must include only letters, digits, and underscore (_) symbols. Blank characters and other non-alphanumeric symbols are not allowed.
- The initial character must be a letter or an underscore.
- It is not case-sensitive, so "**my_Var**" and "**MY_vaR**" both denote the same identifier.

Note: *If you are using the **C Compiler Version** of 4GL, there is a chance that only the first seven characters of a 4GL identifier are recognized by your C compiler. If this is the case with your C compiler, or if you want your application to be portable to all C compilers, keep the first seven characters unique among similar program entities that have the same scope of reference. (Scope of reference is explained later in this section.)*

Unexpected results may result if you declare as an identifier some keywords of SQL, of the C or C++ languages, or of the operating system or network. (Appendix H lists some keywords and predefined identifiers of 4GL that should not be used as identifiers of programmer-defined entities.) If you receive an error message that seems unrelated to the 4GL statement that elicits the error, see if the statement uses a reserved word as an identifier.

These rules resemble the rules for naming SQL identifiers, except that:

- SQL identifiers are typically limited to no more than 18 characters. (But *database* names may be limited to 8, 10, or 14, characters, depending on the database engine and the operating system environment.)
- SQL identifiers *within quoted strings* are lettercase-sensitive.
- You can use *reserved words* as SQL identifiers (but such usage may require qualifiers, can produce errors, and makes your code difficult to read).

If you plan to produce a **Rapid Development Version** of your 4GL application, the total length of the names of all 4GL variables must be less than 32,767.

4GL identifiers can be the same as SQL identifiers, but this may require special attention within the scope of the 4GL name. See the section "Scope and Visibility of SQL Identifiers" on page 2-12 for more information. The *Informix Guide to SQL: Reference* describes SQL identifiers.

NLS

The LC_CTYPE environment variable specifies which predefined set of characters can be legally contained in user-defined names. By specifying a character set other than US ASCII by way of LC_CTYPE, non-ASCII characters such as the ö (o-umlaut) character can be included in identifiers without error. For more information about using NLS, see Appendix E.

Scope of Reference of 4GL Identifiers

Any 4GL identifier can be characterized by its *scope of reference*, sometimes called its *name scope*, or simply its *scope*. A point in the program where an entity can be referenced by its identifier is said to be *in* the scope of reference of that identifier. Conversely, any point in the program where the identifier cannot be recognized is said to be *outside* its scope of reference.

Identifiers of Variables

The scope of reference of a variable is determined by *where* in the **.4gl** source module the DEFINE statement appears that declares the identifier. Identifiers of variables can be *local*, *module*, or *global* in their scope.

- *Local 4GL* variables are declared within a program block. These variables cannot be referenced by statements outside the same program block.
- *Module* variables (sometimes called *modular* or *static*) must be declared outside any MAIN, REPORT, or FUNCTION program block. These identifiers cannot be referenced outside the same **.4gl** module.
- *Global* variables are module variables whose visibility you extend to additional modules. If the GLOBALS … END GLOBALS statement declares variables in one module, you can reference those variables in any other module that includes a corresponding GLOBALS *filename* statement.

Other 4GL Identifiers

Also global in scope are names of predefined entities, and of 4GL windows, forms, reports, and functions. The scope of the identifiers of form entities (like *screen fields, screen arrays, screen records,* or *table alias*es) includes the 4GL statements that are executed while the form is open. Here is a summary of the scope of reference of 4GL identifiers for various types of program entities:

Named Program Entity	Scope of Reference of Identifier
formal function or report argument	Local (to the function or report definition)
4GL function or report	Global
4GL variable	Global, module, or local (based on declaration)
4GL screen field, array, or record	While the form that declares it is displayed
4GL screen form or window	Global (after it has been declared)
4GL statement label	Local (to the program block in which it appears)
4GL table alias	While the form that declares it is displayed

With each named program entity, or *name space*, 4GL identifiers that have the same scope of reference must be unique.

Note: *In the C language, global variables and functions share the same name space. Unless you compile your* 4GL *source code to the* **Rapid Development Version***, a compilation error results if a global variable has the same identifier as a 4GL function or report.*

4GL recognizes *predefined* identifiers that you can reference without declarations or definitions. These include the names of built-in functions and operators like LENGTH( ) or INFIELD( ), constants like TRUE or FALSE, and variables like **status** or **SQLAWARN**. Predefined identifiers are *global* in scope; unless you declare a conflicting identifier, they are visible in any statement.

Scope and Visibility of SQL Identifiers

The scope of an SQL *cursor* or *prepared object* name is from its DECLARE or PREPARE declaration until the module ends, or until a FREE statement specifies that name. (After FREE, subsequent DECLARE or PREPARE statements in the same module cannot reassign the same name, even to an entity that is identical to whatever FREE referenced.) All other SQL identifiers have *global* scope.

Statements cannot reference the name of a global database entity like a *table*, *column*, or *index* after an SQL data definition statement to DROP the entity is executed, or if the database that contains the entity is not open.

If you assign to a 4GL entity the name of an SQL entity, the 4GL name takes precedence within its scope. To avoid ambiguity in DELETE, INSERT, SELECT, and UPDATE statements (and only in these statements), prefix an @ symbol to the name of any *table* or *column* that has the same name as a 4GL variable. Otherwise, only the 4GL identifier is visible in these ambiguous contexts.

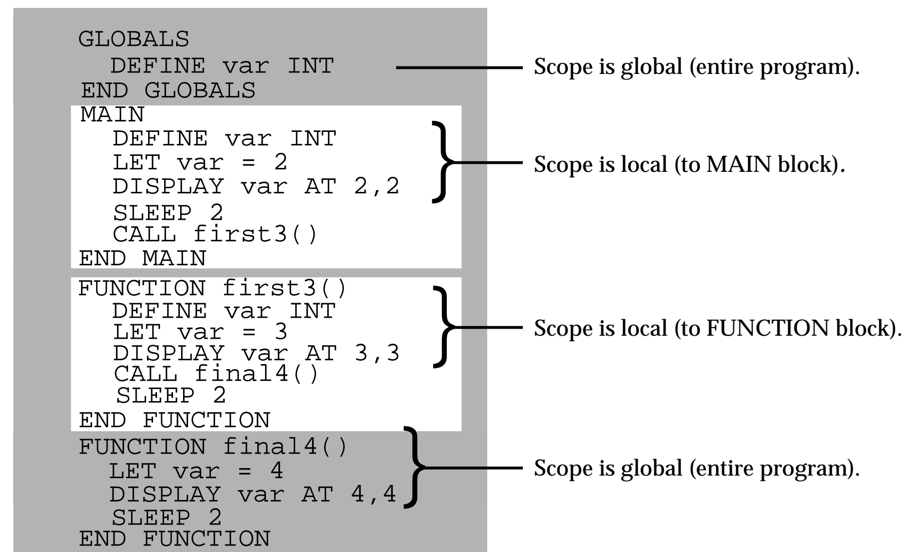
Visibility of Identical Identifiers

A compile-time error occurs if a 4GL program declares the same name for two variables that have the same scope. You can, however, declare the same name for two or more variables that differ in their scope of reference. For example, you can use the same identifier to reference different local variables in different program blocks.

You can also declare the same name for two or more variables whose scopes of reference are different but overlapping. Within their intersection, 4GL interprets the identifier as referencing the entity whose scope is *smaller*. Within the smaller scope of reference, the variable whose scope is a superset of the other is not visible.

Non-Unique Global and Local Variables

If a *local* variable has the same identifier as a *global* variable, the local variable takes precedence inside the block in which it is declared. Elsewhere in the program, the identifier references the global variable, as illustrated in the following INFORMIX-4GL program:



The shaded area indicates the scope of the *global* identifier called **var**. This is superseded in the MAIN and in the first FUNCTION program blocks by the identifiers of *local* variables that have the same name. Only the last DISPLAY statement references the global variable; the first two display local variables.

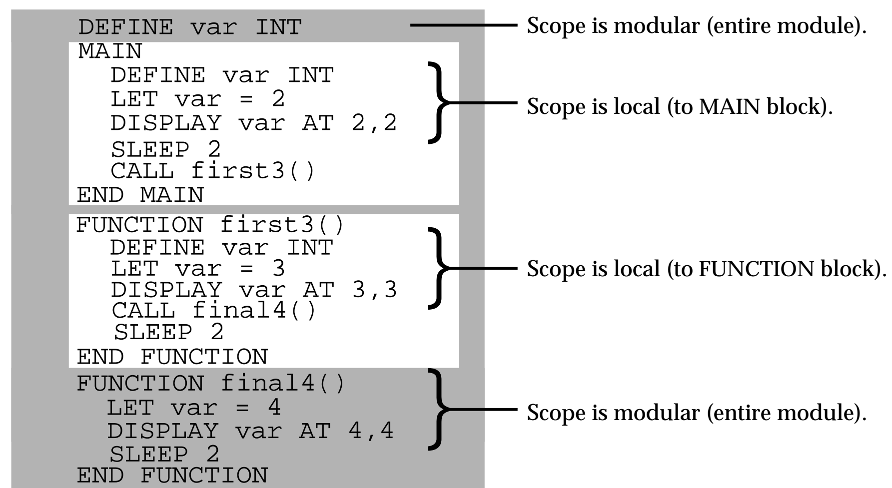
Non-Unique Global and Module Variables

The identifier of a module variable can be the same as a global identifier that is declared in a different module. Within the module where it is declared, the *module* identifier takes precedence over the *global* identifier. Statements within that module cannot reference the global identifier.

The identifier of a module 4GL variable cannot be the same as the name of a global variable that is declared in the same module.

Non-Unique Module and Local Variables

If a *local* variable has the same identifier as a *module* variable, then the local identifier takes precedence inside the program block in which it is declared. Elsewhere in the same source code module, the name references the module variable, as illustrated in the following example:



The shaded area indicates the scope of the *module* variable called **var**. This is superseded in the MAIN block and in the first FUNCTION program block by the identifiers of *local* variables called **var**. The first two DISPLAY statements show values of local variables; the last displays the module variable.

Within the portion of a program where more than one variable has the same identifier, INFORMIX-4GL gives precedence to a module identifier over a global one, and to a local identifier over one with any other scope. Assign unique names to variables, if you wish to avoid masking part of the scope of non-unique module identifiers.

Interacting with Users

You can use INFORMIX-4GL to create applications that interact with users in the following ways:

- Menus (page 2-15)
- Screen forms (page 2-17)
- 4GL windows (page 2-19)
- Help messages (page 2-22)
- Reports based on data retrieved from an SQL database (page 6-3)

Ring Menus

You can use the MENU statement of 4GL to create and display a *ring menu* of command options, so that users can perform the tasks that you specify. The menu of a typical 4GL program, for example, might look like this:

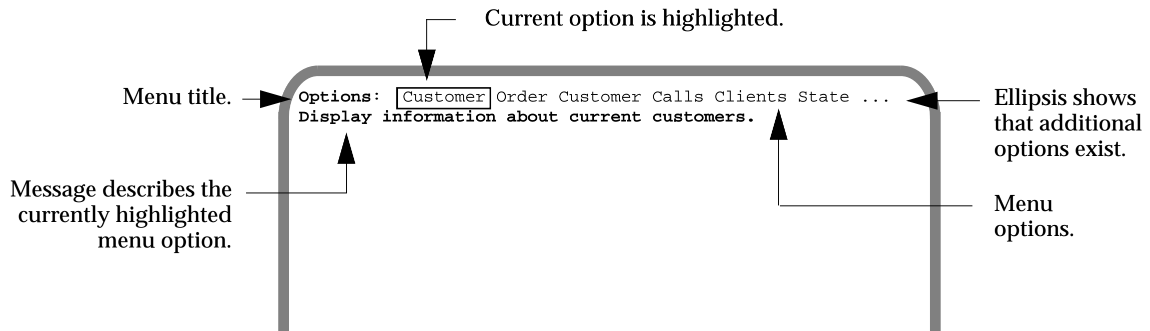


Figure 2-1 ***The format of a typical 4GL ring menu***

If a menu has more options than can fit on one line, ellipsis (. . .) symbols automatically indicate that more options appear on another page of the menu. In this example, the ellipsis indicates that additional menu options are on one or more pages to the right. Similarly, an ellipsis on the left means that additional menu options are on one or more menu pages to the left.

The user can scroll to the right to display the next page of options by using the Right Arrow key or Spacebar, or scroll to the left by using the Left Arrow key.

Option names can include embedded blank characters. By default, an option is chosen when the user types its initial character, but you can specify additional activation keys. Different menus can have the same option names.

You can nest MENU statements within other MENU statements, so that the menus form a hierarchy. A nested MENU statement can appear directly within a menu control block, or else in a function that is called directly or indirectly when the user chooses an option of the enclosing menu.

Selecting Menu Options

By pressing the RETURN key, the user can select the menu option that is currently highlighted in reverse video. In the previous example, **Customer** would be selected. The highlight that indicates the current option is called the *menu cursor.*

This is called a *ring menu* because the menu cursor behaves as if the list of options were cyclic: if the user moves the cursor to the right, past the last option, then the *first* option is highlighted. Similarly, moving the menu cursor to the left, past the first option, highlights the *last* option.

Pressing the key that matches the initial character of a menu option, such as `O` (for **Order**) in the preceding illustration, selects the corresponding option.

All other options are disabled until the associated COMMAND block completes its execution. Disabled menu options cannot be selected.

Ambiguous Keyboard Selections

If the user makes an ambiguous menu option selection (for example, by typing `C` in the 4GL menu containing **Customer**, **Customer Calls**, and **Clients** in the previous example, **4GL** clears the second line of the menu and prompts the user to clarify the choice. **4GL** displays each keystroke, followed by the names of the menu options that begin with the typed letters. When **4GL** identifies a unique option, it closes this prompt line and executes the statements associated with the selected menu option. The backspace key undoes the most recently typed key.

Hidden Options and Invisible Options

You can suppress the display of any subset of the menu options, disabling these *hidden options.* The section "The HIDE OPTION and SHOW OPTION Keywords" on page 3-203 describes how the MENU statement can programmatically control whether or not a menu option is hidden or accessible.

Menus can also include *invisible options.* An invisible option does not appear in the menu, but it performs the specified actions when the user presses the activation key. See page 3-200 for a description of how to create invisible menu options.

Disabled Menus

Menus themselves are not always accessible. During screen interaction statements like INPUT, CONSTRUCT, INPUT ARRAY, or DISPLAY ARRAY, errors would be likely to result if the user could interrupt the interaction with menu choices. **4GL** prevents these errors by disabling the entire menu during the execution of these statements. The menu does not change its appearance when it is disabled.

Reserved Lines for Menus

The first line (called the *Menu line*) lists a *title* and *options* of the menu. A *menu cursor* (a double border) highlights the current option. For each option, a *menu control block* (page 3-194) specifies statements to execute if the user chooses the option.

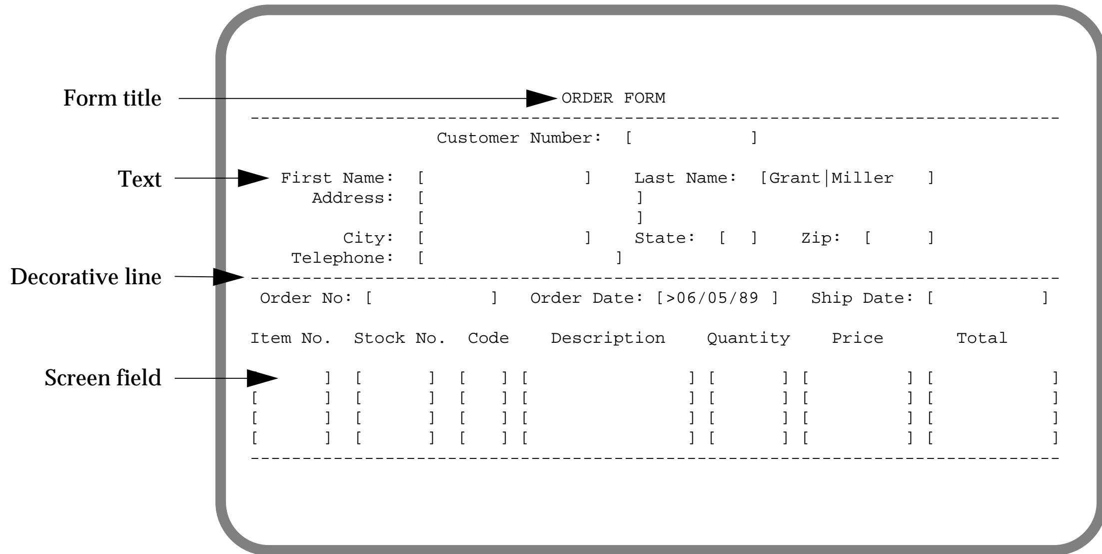
The next line (called the *Menu help line*) displays a prompt for the currently highlighted option. If the user moves the menu cursor to another option, this prompt is replaced by one for the new current option.

Screen Forms

A *screen form* is a 4GL display in which the user can view, enter, or edit data. The following elements can appear in a 4GL screen form:

- *Fields*: Areas (also called *form fields* or *screen fields*) where the user enters or edits data, or where the 4GL program displays values (page 5-14).
- *Field delimiters*: Fields are usually enclosed by square brackets (page 5-14).
- *Screen records*: Logically related sets of fields (page 5-63).
- *Screen arrays*: Scrollable arrays of fields or of screen records (page 5-66).
- *Decorative rectangles*: These can enclose all or part of the form (page 4-56).

- *Text:* Anything else in the form is called *text*. This always appears while the form is visible. Text can include display labels to identify fields, titles for the form or for its parts, or other ornaments, as in this example:



Chapter 5 describes how to create screen forms.

Visual Cursors

4GL marks the user's current location (if any) in the current menu, form, or field with a *visual cursor.* Usually, each of these is simply called "the cursor":

- Menu cursor: Reverse video marks the option that the RETURN key can choose.
- Field cursor: This vertical (|) bar marks the current character position in the current field.

Field Attributes

Several 4GL statements can set display attributes (page 3-290). A form specification file can also specify *field attributes.* These optional descriptors can control the display when the cursor is in the field, or can supply or restrict field values during data entry. Field attributes can have effects like these:

- Control *cursor movement* among fields.
- Set *validation* and *default value* field attributes.

- Set *formatting* attributes, or automatically invoke a *multiple-line editor* for character data, or an *external editor* to view or modify TEXT or BYTE data.
- Set video display *color* or *intensity* attributes.

Reserved Lines

On the 4GL screen, certain lines are reserved for output from specific 4GL statements or from other sources. By default, these *reserved lines* appear in the following positions on the **4GL** screen:

Menu line Line 1 displays the *menu title* and *options* list from MENU.

Prompt line Line 1 also displays text specified by the PROMPT statement.

Menu help line Line 2 displays text describing MENU options. You cannot reposition this line independently of the Menu line.

Message line Line 2 also displays text from the MESSAGE statement. You *can* reposition this line with the OPTIONS statement.

Form line Line 3 begins a form display when DISPLAY FORM executes.

Comment line The (*last - 1)* line of the 4GL screen (or *last* line in a **4GL** window) displays COMMENTS attribute messages.

Error line The *last* line of the 4GL screen displays output from the ERROR statement.

If you display the form in a named 4GL window (page 2-19), these default values apply to that window, rather than to the 4GL screen, except for the Error line. (The position of the Error line is defined relative to the entire screen, rather than to the 4GL window.)

The OPTIONS statement (page 3-228) can change these default positions for all the **4GL** windows of your application. The OPEN WINDOW statement (page 3-219) can reposition all of these reserved lines (except the Error line) within the specified 4GL window.

4GL Windows

A 4GL *window* is a named rectangular area on the 4GL screen. When a **4GL** program starts, the entire 4GL screen is the current window. In **4GL** statements that can reference windows, the name of this default window is SCREEN. The OPEN WINDOW statement (page 3-219) can create additional 4GL windows, and can specify the dimensions and attributes of each window. In DBMS applications that perform various tasks, displaying distinct activities in different 4GL windows may make it easier for users to operate your program.

Every 4GL window can display no more than one 4GL form. The CURRENT WINDOW statement can transfer control from one 4GL window to another.

The Current Window

INFORMIX-4GL maintains a list of all the open 4GL windows, called the *window stack*. When you open a new 4GL window, it is added to the top of this stack. The window at the top of the stack is called the *current window*.

The current 4GL window is always completely visible, and can obscure all or part of any inactive windows. When you specify a current window, 4GL adjusts the window stack by moving the new current window to the top, and closing the gap in the stack left by this window. When you close a window, 4GL removes that window from the window stack. The topmost window among those that remain on the screen becomes the current window, if the 4GL window that was closed was the current window. All this takes place within the 4GL screen.

All input and output is done in the current window. If that window contains a screen form, the form becomes the *current form*. The DISPLAY ARRAY, INPUT, INPUT ARRAY, and MENU statements all run in the current window. If a user displays a form in another window from within one of these statements (for example, by activating an ON KEY block), the window containing the new form becomes the current window. When the enclosing statement resumes execution, the original window is restored as the current window.

Programs with multiple windows may need to switch to a different current window unconditionally, so that input and output occur in the appropriate window. The CURRENT WINDOW statement (page 3-56) makes a specified window (or SCREEN) the current window. When a window becomes the current window, 4GL restores its values for the positions of the Prompt, Menu, Message, Form, and Comment lines.

On-Line Help

INFORMIX-4GL includes two distinct facilities for displaying Help messages:

- **Development Help**

  The developer can request Help from the Programmer's Environment regarding features of the INFORMIX-4GL language. Press the Help key to display help on the currently selected menu option.

- **Run-Time Help**

  The user of a 4GL application can display programmer-defined Help messages.

Run-time Help is displayed when the user presses a designated Help key. The 4GL statements that can include a HELP clause are these:

- CONSTRUCT (during a query-by-example)
- INPUT and INPUT ARRAY (during data entry)
- MENU (for each menu option)
- PROMPT (when the user must supply keyboard input)

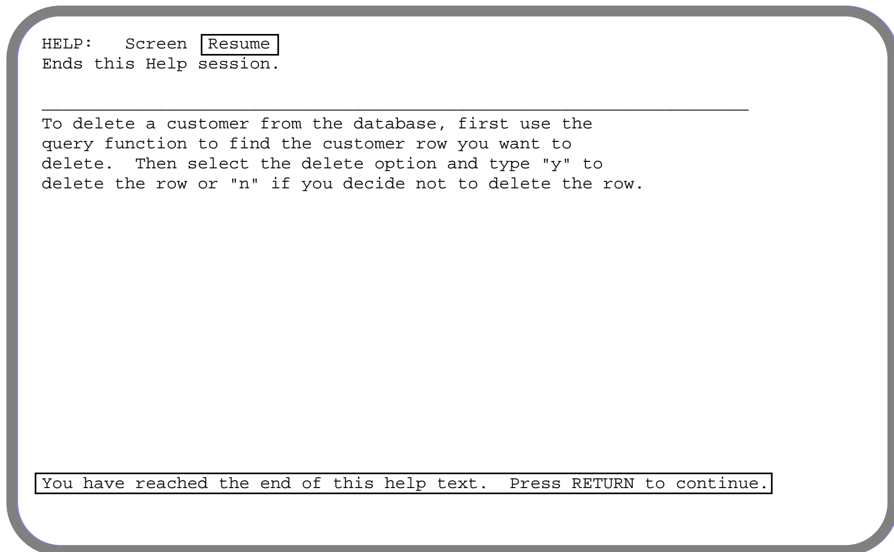
The HELP clause specifies a single Help message for the entire 4GL statement. To provide field-level Help in these interactive statements, you can use an ON KEY clause with the INFIELD( ) operator (page 4-69) and the SHOW_HELP( ) function (page 4-81).

The Help Key and the Message Compiler

By default, CONTROL-W acts as the Help key. To specify a non-default Help key, or to identify a file that contains Help messages, use the OPTIONS statement. If you specify a file of Help messages, 4GL displays the messages in the Help window.

The Help Window

Here is an example of a typical Help window display:



When the user presses the Help key, Help appears in a *Help window*. The 4GL screen is hidden while this window is open.

The Help window has a 4GL ring menu containing **Screen** and **Resume** menu options. **Screen** display the next page of Help text. **Resume** closes the Help window and redisplays the 4GL screen.

You must create these Help messages and store them in an ASCII file. Each message begins with a unique whole number that has an absolute value no greater than 32,767 and is prefixed by a period (**.**) symbol. Statements of 4GL can reference a Help message by specifying its number in a HELP clause. You create run-time messages from an ASCII source file by using the **mkmessage** utility. For more information about creating Help messages, see "The mkmessage Utility" on page B-2.

The Help window persists until the user closes it. The user can dismiss the Help window by using the **Resume** menu option or by pressing RETURN.

Exception Handling

INFORMIX-4GL provides facilities for issuing compile-time and link-time detection of compiler errors, and for run-time detection and handling of *exceptional conditions*. These can include any of the following:

- SQL errors, warnings, or NOTFOUND conditions that the database engine detects, and that 4GL automatically records in the **SQLCA** area.
- Interrupt, Quit, or other signals from the user or from other sources.
- Run-time errors that **4GL** issues.

You can use DEFER and WHENEVER statements to trap and handle errors.The WHENEVER statement can control the processing of exceptional conditions of several kinds: SQL warnings, SQL end of data, errors in SQL or screen operations, or errors in expressions of all kinds.

The DEFER statement can instruct **4GL** not to terminate the program when the Interrupt or Quit key is pressed. The DEFER statement has dynamic scope, as opposed to the lexical scope of WHENEVER. When Quit or Interrupt is deferred, it is ignored globally (that is, in all modules).

See the descriptions of DEFER and WHENEVER in Chapter 3 for details of how to use these statements to handle interrupts, errors, and warnings.

Error Handling with SQLCA

Proper database management requires that all logical sequences of statements that modify the database continue successfully to completion. Suppose, for example, that an UPDATE of a customer account shows a reduction of $100.00 in the payable balance, but for some reason the next step, an UPDATE of the cash balance, fails; now your books are out of balance. It is prudent to verify that every SQL statement executes as you anticipated.

INFORMIX-4GL provides two ways to do this:

- The global variable **status** that can indicate errors both from interactive statements and from SQL statements.
- A global record **SQLCA** that allows you to test the success of SQL statements, and to obtain other information about actions by the engine.

Compared to **status**, the **SQLCA.SQLCCODE** variable is typically easier to use for monitoring the success or failure SQL statements, because it ignores exceptional conditions that may be encountered in other 4GL statements.

INFORMIX-4GL returns a result code into the **SQLCA** record after executing every SQL statement except DECLARE. The **SQLCA** record has this structure:

```
DEFINE  SQLCA  RECORD
    SQLCODE INTEGER,
    SQLERRM CHAR(71),
    SQLERRP CHAR(8),
    SQLERRD ARRAY [6]  OF  INTEGER,
    SQLAWARN CHAR (8)
END  RECORD
```

SQLCODE indicates the result of executing an SQL statement. It is set to zero for a successful execution of most statements and to NOTFOUND (= 100) for a successfully executed query that returns zero rows or for a FETCH that seeks beyond the end of an active set.

SQLCODE is negative after an unsuccessful execution.

At run time, **INFORMIX-4GL** sets the global variable **status** equal to **SQLCODE** after each SQL statement. (See also the description of the ANY keyword of WHENEVER on page 2-26.) The *Informix Error Messages, Version 6.0* manual provides explanations of SQL and **4GL** error codes.

SQLERRM is not used at this time.

SQLERRP is not used at this time.

SQLERRD an array of six variables of data type INTEGER

- **SQLERRD[1]** is not used at this time.
- **SQLERRD[2]** is a SERIAL value returned or **ISAM** error code.
- **SQLERRD[3]** is the number of rows processed.
- **SQLERRD[4]** is the estimated CPU cost for query.
- **SQLERRD[5]** is the offset of error into the SQL statement.
- **SQLERRD[6]** is the ROWID of last row.

SQLAWARN is a character string of length eight whose individual characters signal various warning conditions (as opposed to errors) following the execution of an SQL statement. The characters are blank if no problems were detected.

SQLAWARN[1] is set to `W` if any of the other warning characters has been set to `W`. If **SQLAWARN[1]** is blank, you do not have to check the remaining warning characters.

SQLAWARN[2] is set to `W` if one or more data items was truncated to fit into a CHAR program variable, or if a DATABASE statement selected a database with transactions.

SQLAWARN[3] is set to `W` if an aggregate like SUM( ), AVG( ), MAX( ), or MIN( ) encountered a NULL value in its evaluation, or if the DATABASE statement specified an ANSI-compliant database.

SQLAWARN[4] is set to `W` if a DATABASE statement selected an **INFORMIX-OnLine** database, or when the number of items in the *select-list* of a SELECT clause is not the same as the number of program variables in the INTO clause. The number of values returned by 4GL is the smaller of these two numbers.

SQLAWARN[5] is set to `W` if float-to-decimal conversion is used.

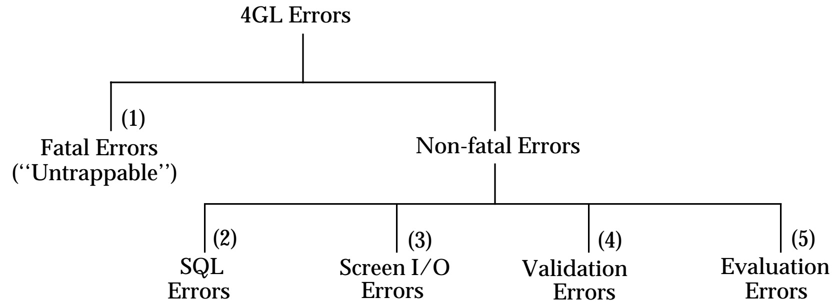
SQLAWARN[6] is set to `W` when your program executes an **INFORMIX-4GL** extension to ANSI standard syntax, and the DBANSIWARN environment variable is set.

SQLAWARN[7] is not used at present.

SQLAWARN[8] is not used at present.

A Taxonomy of Run-Time Errors

The WHENEVER statement classifies 4GL run-time errors into five disjunct categories:



The WHENEVER statement cannot trap *fatal* errors, the category marked (1). By definition, a fatal error terminates the program immediately with an error message, regardless of any WHENEVER directive.

To trap errors of types (2), (3), and (4), specify WHENEVER ERROR. However, if you want to additionally trap errors of type (5), you need to specify WHENEVER ANY ERROR. (These are arithmetic, Boolean, or conversion errors that can occur when 4GL evaluates an expression.)

For a list of all the 4GL run-time errors, see *Informix Error Messages, Version 6.0*.

Fatal Run-Time Errors

In the context of WHENEVER ERROR or WHENEVER ANY ERROR directives, a "fatal" error means an *untrappable* error. If you specify WHENEVER ERROR STOP, then any error terminates the program, but if you specify WHENEVER ERROR CONTINUE, 4GL attempts to continue execution after most errors.

The following, however, are fatal run-time errors in category (1):

-1318 A parameter count mismatch has occurred between the calling function and the called function.

-1319 The 4GL program has run out of run-time data space memory.

-1322 A report output file cannot be opened.

-1323 A report output pipe cannot be opened.

-1324 A report output file cannot be written to.

-1326 An array variable has been referenced outside of its specified dimensions.

-1332 A character variable has referenced subscripts that are out of range.

-1335 A report is accepting output or being finished before it has been started.

-1340 The error log has not been started.

-1362 4GL run-time stack violation.

-1379 Report functions may not be called directly. Please use the OUTPUT TO REPORT statement.

-4339 4GL has run out of data space memory.

-4392 The 4GL compiler has run out of data space memory to contain the 4GL program symbols. If the program module is very large, dividing it into separate modules may alleviate the situation.

-4508 PRINT FILE error - cannot open file "*filename*" for reading.

-4517 Strings of length > 512 cannot be returned from function calls.

-4518 The 4GL program cannot allocate any more space for temporary string storage.

-4623 Memory allocation error.

Note: *This list may not be complete, and is likely to change in future releases, or even between the time when this manual is written and your current release. If you need detailed information about untrappable errors, examine the release notes in* **$INFORMIXDIR/release/TOOLS_6.0** *to see if new fatal errors exist.*

Chapter

3

INFORMIX-4GL Statements

Chapter Overview 11

The 4GL Statement Set 11
- Types of SQL Statements 11
- Other Types of 4GL Statements 13

Statement Descriptions 15
- CALL 16
  - Arguments 17
  - The RETURNING Clause 19
  - Restrictions on Returned Character Strings 20
  - Invoking a Function Without CALL 20
- CASE 21
  - The WHEN Blocks 22
  - The OTHERWISE Block 23
  - The EXIT CASE Statement 24
  - The END CASE Keywords 24
- CLEAR 26
  - The CLEAR FORM Option 26
  - The CLEAR WINDOW Option 26
  - The CLEAR WINDOW SCREEN Option 27
  - The CLEAR SCREEN Option 27
  - The CLEAR field Option 27
- CLOSE FORM 29
- CLOSE WINDOW 30

CONSTRUCT 31
- The CONSTRUCT Variable Clause 33
- The ATTRIBUTE Clause 37
- The HELP Clause 37
- The CONSTRUCT Form Management Blocks 38
- The NEXT FIELD Clause 44
- The CONTINUE CONSTRUCT Statement 46
- The EXIT CONSTRUCT Statement 47
- The END CONSTRUCT Keywords 47
- Using Built-In Functions and Operators 47
- Query by Example 48
- Positioning the Screen Cursor 50
- Editing During a CONSTRUCT Statement 52
- Completing a Query 52

CONTINUE 55

CURRENT WINDOW 56

DATABASE 58
- The Database Specification 59
- The Default Database at Compile Time 59
- The Current Database at Run Time 60
- The EXCLUSIVE Keyword 61
- Testing SQLCA.SQLAWARN 61

DEFER 62
- Interrupting Screen Interaction Statements 62
- Interrupting SQL Statements 64

DEFINE 65
- The Context of DEFINE Declarations 65
- Declaring the Names and Data Types of Variables 67
- Variables of Simple Data Types 68
- Variables of Large Data Types 70
- Variables of Structured Data Types 70

DISPLAY 74
- Sending Output to the Line Mode Overlay 76
- Sending Output to the Current 4GL Window 77
- Sending Output to a Screen Form 80
- The ATTRIBUTE Clause 83
- Displaying Numeric and Monetary Values 83

DISPLAY ARRAY 85
The ATTRIBUTE Clause 86
The ON KEY Blocks 87
The EXIT DISPLAY Statement 89
The END DISPLAY Keywords 89
Using Built-In Functions and Operators 90
Scrolling During the DISPLAY ARRAY Statement 91
Completing the DISPLAY ARRAY Statement 92
DISPLAY FORM 93
Form Attributes 93
Reserved Lines 93
END 95
ERROR 96
The ATTRIBUTE Clause 97
System Error Messages 97
EXIT 98
Leaving a Control Structure 98
Leaving the Program 99
FINISH REPORT 100
FOR 102
The TO Clause 102
The STEP Clause 103
The CONTINUE FOR Statement 103
The EXIT FOR Statement 104
The END FOR Keywords 104
Databases with Transactions 104
FOREACH 105
Cursor Names 106
The INTO Clause 107
The FOREACH Statement Block 108
The END FOREACH Keywords 109
FUNCTION 111
The Prototype of the Function 112
The FUNCTION Program Block 113
Data Type Declarations 113
The Function as a Local Scope of Reference 114
Executable Statements 115
Returning Values to the Calling Routine 115
The END FUNCTION Keywords 116
GLOBALS 117
Declaring and Exporting Global Variables 117
Importing Global Variables 119
GOTO 122

IF 124
INITIALIZE 125
- The LIKE Clause 126
- The TO NULL Clause 127

INPUT 128
- The Binding Clause 129
- The ATTRIBUTE Clause 133
- The HELP Clause 133
- The INPUT Form Management Blocks 134
- The CONTINUE INPUT Statement 143
- The EXIT INPUT Statement 143
- The END INPUT Keywords 144
- Using Built-In Functions and Operators 144
- Keyboard Interaction 146
- Cursor Movement in Simple Fields 146
- Multiple-Segment Fields 147
- Using Large Data Types 149
- Completing the INPUT Statement 150

INPUT ARRAY 152
- The Binding Clause 153
- The ATTRIBUTE Clause 155
- The HELP Clause 155
- The INPUT ARRAY Form Management Blocks 156
- The CONTINUE INPUT Statement 169
- The EXIT INPUT Statement 169
- The END INPUT Keywords 169
- Using Built-In Functions and Operators 170
- Keyboard Interaction 172
- Using Large Data Types 174
- Completing the INPUT ARRAY Statement 174

LABEL 177
LET 178
LOAD 181
- The Input File 182
- The DELIMITER Clause 184
- The INSERT Clause 184

LOCATE 186
- The List of Large Variables 187
- The IN MEMORY Option 187
- The IN FILE Option 188
- Passing Large Variables to Functions 189
- Freeing the Storage Allocated to a Large Variable 190

MAIN 191
    The END MAIN Keywords 191
    Variables Declared in the MAIN Statement 192
MENU 193
    The MENU Control Blocks 194
    Invisible Menu Options 200
    The CONTINUE MENU Statement 201
    The EXIT MENU Statement 202
    The NEXT OPTION Clause 203
    The HIDE OPTION and SHOW OPTION Keywords 203
    Nested MENU Statements 205
    The END MENU Keywords 205
    Identifiers in the MENU Statement 205
    Choosing a Menu Option 206
    Scrolling the Menu Options 208
    Completing the MENU Statement 209
MESSAGE 213
    The Message Line 213
    The ATTRIBUTE Clause 214
NEED 216
OPEN FORM 217
    The Form Name 217
    Specifying a Filename 217
    Displaying a Form in a 4GL Window 218
OPEN WINDOW 219
    The 4GL Window Stack 220
    The AT Clause 220
    The WITH ROWS, COLUMNS Clause 221
    The WITH FORM Clause 221
    The OPEN WINDOW ATTRIBUTE Clause 222
OPTIONS 228
    Features Controlled by OPTIONS Clauses 229
    Positioning Reserved Lines 231
    Cursor Movement in Interactive Statements 232
    The OPTIONS ATTRIBUTE Clause 233
    The HELP FILE Option 234
    Assigning Logical Keys 234
    Interrupting SQL Statements 235
OUTPUT TO REPORT 242
PAUSE 244

PREPARE 245
  Statement Identifier 246
  Releasing a Statement Identifier 246
  Statement Text 246
  Preparing Statements in 4GL 247
  Using Parameters in Prepared Statements 250
  Preparing Statements with SQL Identifiers 251
  Preparing Sequences of Multiple SQL Statements 252
  Using Prepared Statements for Efficiency 253
PRINT 254
PROMPT 255
  The PROMPT String 256
  The Response Variable 256
  The FOR Clause 257
  The ATTRIBUTE Clauses 257
  The HELP Clause 258
  The ON KEY Blocks 258
  The END PROMPT Keywords 259
REPORT 260
  The Report Prototype 261
  The Report Program Block 261
  The END REPORT Keywords 262
  Two-Pass Reports 262
RETURN 263
  The Data Types of Returned Values 264
RUN 265
  The RETURNING Clause 266
  The WITHOUT WAITING Clause 267
SCROLL 268
SKIP 269
SLEEP 270
START REPORT 271
  The TO Clause 271
UNLOAD 274
  The Output File 274
  The DELIMITER Clause 276
VALIDATE 278
  The LIKE Clause 279
  The syscolval Table 280

WHENEVER 281
The Scope of the WHENEVER Statement 282
The ERROR Condition 282
The ANY ERROR Condition 283
The NOT FOUND Condition 283
The WARNING Condition 284
The GOTO Option 284
The CALL Option 285
The CONTINUE Option 285
The STOP Option 285
WHILE 287
The CONTINUE WHILE Statement 288
The EXIT WHILE Statement 288
The END WHILE Keywords 288

Statement Segments 289
ATTRIBUTE 290
Color and Monochrome Attributes 291
Precedence of Attributes 292
Data Types of 4GL 293
The Simple Data Types 294
Number Data Types 295
Time Data Types 295
Character Data Types 296
The Structured Data Types 296
The Large Data Types 296
Descriptions of the 4GL Data Types 296
ARRAY 297
BYTE 298
CHAR 299
CHARACTER 300
DATE 300
DATETIME 300
DEC 304
DECIMAL 304
DOUBLE PRECISION 305
FLOAT 305
INT 306
INTEGER 306
INTERVAL 307
MONEY 312
NUMERIC 313
REAL 313
RECORD 313

SMALLFLOAT 315
SMALLINT 316
TEXT 317
VARCHAR 318
Data Type Conversion 319
- Converting from Number to Number 319
- Converting Numbers in Arithmetic Operations 320
- Converting Between DATE and DATETIME 321
- Converting CHAR to DATETIME or INTERVAL Data Types 322
- Converting Between Number and Character Data Types 323

Summary of Compatible 4GL Data Types 324
- Notes on Automatic Data Type Conversion 325

Expressions of 4GL 326
Components of 4GL Expressions 327
- Parentheses in 4GL Expressions 327
- Operators in 4GL Expressions 327
- Operands in 4GL Expressions 331
- Named Values as Operands 331
- Function Calls as Operands 332
- Expressions as Operands 332

4GL Boolean Expressions 333
- Logical Operators 333
- Boolean Comparisons 334
- Data Type Compatibility 337
- Evaluating 4GL Boolean Expressions 338

Integer Expressions 338
- Binary Arithmetic Operators 339
- Unary Arithmetic Operators 340
- Literal Integers 340

Number Expressions 341
- Arithmetic Operators 341
- Literal Numbers 342

Character Expressions 343
- Arrays and Substrings 344
- String Operators 344
- Non-Printable Characters 345

Time Expressions 347
Numeric Date 349
DATETIME Qualifier 349
DATETIME Literal 351
INTERVAL Qualifier 353
INTERVAL Literal 355
Arithmetic Operations on Time Values 356
Relational Operators and Time Values 358
Field Clause 359
Table Qualifiers 361
Owner Naming 361
Database References 362
THRU or THROUGH Keywords and .* Notation 363

Chapter Overview

This chapter describes the **INFORMIX-4GL** statements. Information about their syntax and usage is presented in three sections:

- The statement set of **4GL**, classified within functional categories.
- Descriptions of the non-**SQL** statements of **4GL**, in alphabetic order.
- Other **4GL** elements, called *segments*, that can appear within statements.

The 4GL Statement Set

INFORMIX-4GL supports the SQL language, but it is sometimes convenient to distinguish between SQL statements and other **4GL** statements:

- SQL statements operate on tables and their columns in a database.
- **4GL** statements operate on variables in memory.

Types of SQL Statements

The SQL statements of **4GL** can be divided into these functional categories:

- Data definition statements
- Data manipulation statements
- Cursor manipulation statements
- Dynamic-management statements
- Query optimization information statements
- Data access statements
- Data integrity statements
- Stored-procedure statements
- **INFORMIX-OnLine/Optical** statements

The SQL statements in each of these functional categories are listed on the following pages.

***Note:** For syntax and usage information about most SQL statements, see the Informix Guide to SQL: Reference. If the statement is preceded by a 6.0 icon in the following lists, see the Informix Guide to SQL: Syntax. To use the SQL statements identified by the 6.0 icon in a* **4GL** *program, you must prepare the statement; for details, see* *page 3-247**.*

SQL Data Definition Statements

- ALTER INDEX
- ALTER TABLE
- CLOSE DATABASE
- CREATE DATABASE
- CREATE INDEX
- 6.0 CREATE PROCEDURE
- 6.0 CREATE PROCEDURE FROM
- 6.0 CREATE SCHEMA
- CREATE SYNONYM
- CREATE TABLE
- 6.0 CREATE TRIGGER
- CREATE VIEW
- DATABASE
- DROP DATABASE
- DROP INDEX
- 6.0 DROP PROCEDURE
- DROP SYNONYM
- DROP TABLE
- 6.0 DROP TRIGGER
- DROP VIEW
- RENAME COLUMN
- RENAME TABLE

SQL Data Manipulation Statements

- INSERT
- DELETE
- LOAD
- SELECT
- UNLOAD
- UPDATE

SQL Cursor Manipulation Statements

- CLOSE
- DECLARE
- FETCH
- FLUSH
- OPEN
- PUT

SQL Dynamic-Management Statements

- EXECUTE
- FREE
- PREPARE

SQL Query Optimization Information Statements

- SET EXPLAIN
- 6.0 SET OPTIMIZATION
- UPDATE STATISTICS

SQL Data Access Statements

- GRANT
- LOCK TABLE
- REVOKE
- OL SET ISOLATION
- OL SET LOCK MODE
- UNLOCK TABLE

SQL Data Integrity Statements

	BEGIN WORK		ROLLBACK WORK
	COMMIT WORK	SE	ROLLFORWARD DATABASE
SE	CREATE AUDIT	OL	SET CONSTRAINTS
SE	DROP AUDIT	OL	SET LOG
SE	RECOVER TABLE	SE	START DATABASE

SQL Stored-Procedure Statements

6.0	EXECUTE PROCEDURE	6.0	SET DEBUG FILE TO

SQL INFORMIX-OnLine/Optical Statements

6.0	ALTER OPTICAL CLUSTER
6.0	CREATE OPTICAL CLUSTER
6.0	DROP OPTICAL CLUSTER
6.0	RELEASE
6.0	RESERVE
6.0	SET MOUNTING TIMING

Note: *INFORMIX-OnLine/Optical statements are shown and described in INFORMIX-OnLine/Optical User Manual.*

Other Types of 4GL Statements

Six other types of **4GL** statements are available. (These are sometimes called simply "**4GL** statements," to distinguish them from SQL statements.)

- Definition and declaration statements
- Program flow control statements
- Compiler directives
- Storage manipulation statements
- Report execution statements
- Screen interaction statements

4GL Definition and Declaration Statements

DEFINE	MAIN
FUNCTION	REPORT

4GL Program Flow Control Statements

CALL	GOTO
CASE	IF
CONTINUE	LABEL
DATABASE	OUTPUT TO REPORT
END	RETURN
EXIT	RUN
FINISH REPORT	START REPORT
FOR	WHILE
FOREACH	

4GL Compiler Directives

DATABASE	GLOBALS
DEFER	WHENEVER

4GL Storage Manipulation Statements

INITIALIZE	LOCATE
LET	VALIDATE

4GL Screen Interaction Statements

CLEAR	INPUT
CLOSE FORM	INPUT ARRAY
CLOSE WINDOW	MENU
CONSTRUCT	MESSAGE
CURRENT WINDOW	OPEN FORM
DISPLAY	OPEN WINDOW
DISPLAY ARRAY	OPTIONS
DISPLAY FORM	PROMPT
ERROR	SCROLL
	SLEEP

4GL Report Execution Statements

NEED	PRINT
PAUSE	SKIP

Most **4GL** statements are not sensitive to whether **INFORMIX-SE** or the **OnLine** engine supports the application. Only the **OnLine** engine, however, can store values in BYTE, TEXT, or VARCHAR columns, or can accept *database:* or *database@system:* as qualifiers to names of tables, views, or synonyms.

Statement Descriptions

The sections that follow describe in alphabetical order the **4GL** statements that are not SQL statements, as well as the SQL statements DATABASE, LOAD, PREPARE, and UNLOAD. Each statement description includes the following elements:

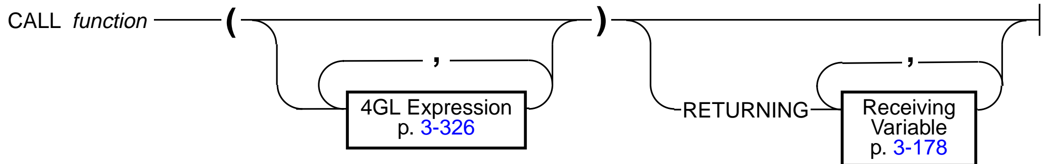
- The name and a terse summary of the effect of the **4GL** statement.
- A syntax diagram.
- Notes on usage, typically arranged by keyword options or by other syntax elements.

If a description is longer than a few pages, a table identifies the major topical headings and their page numbers.

A list of related statements concludes most of these statement descriptions.

CALL

The CALL statement invokes a specified function.



function is the name of the function to be executed.

Usage

You can use the CALL statement to invoke functions from a 4GL application:

- Programmer-defined 4GL functions
- 4GL built-in functions
- C language functions
- **ESQL/C** functions (if you have the INFORMIX-ESQL/C product)

Programmer-defined 4GL functions are defined in FUNCTION statements (page 3-111). These can appear in the same source file as the MAIN statement, or they can be compiled in separate **.4gl** modules (individually, or with other function and report definitions) and linked later to the MAIN program block.

When 4GL encounters a CALL statement at run time, it locates the specified FUNCTION program block and executes its statements in sequence. If the function is not a built-in function, a link-time error occurs unless exactly one definition of that function exists in the modules that comprise the program.

The program block containing the CALL statement is called the *calling routine*. The RETURNING clause can specify the name of one or more variables that *function* returns to the calling routine. This variable (or list of variables) has the same syntax as a *receiving variable* (page 3-178) in the LET statement.

***Note:** Unlike 4GL identifiers, the names of C functions are lettercase-sensitive, and must typically appear in lowercase letters within the function call. For details about how to use C functions in 4GL programs, refer to Appendix C.*

In this example, the CALL statement invokes the **show_menu( )** function:

```
MAIN
    ...
    CALL show_menu()
    ...
END MAIN
FUNCTION show_menu()
    ...
END FUNCTION
```

Chapter 4 provides more information about defining and invoking functions. Sections that follow describe these topics:

Topic	Page
Arguments	3-17
Passing Arguments by Value	3-18
Passing Arguments by Reference	3-18
The RETURNING Clause	3-19
Restrictions on Returned Character Strings	3-20
Invoking a Function without CALL	3-20

Arguments

The *expression list* after the function name specifies values for CALL to pass as *actual arguments* to the function. These actual arguments can be any 4GL expression (page 3-326), if the returned data types are compatible with the corresponding *formal arguments* in the FUNCTION definition. (Parentheses are required around the argument list, even if the list is empty because you do not specify any arguments.)

For example, the following program fragment passes the current values of **p_customer.fname** and **p_customer.lname** to the **print_name( )** function:

```
MAIN
    ...
    CALL print_name(p_customer.fname, p_customer.lname)
    ...
END MAIN
FUNCTION print_name(fname, lname)
    DEFINE fname, lname CHAR(15)
    ...
END FUNCTION
```

When passing arguments to a function, keep the following in mind:

- Values in the argument list must correspond in number and in position to the formal arguments that were specified in the FUNCTION statement.
- Data types of values must be compatible (page 3-324), but need not be identical, to those of the formal arguments in the FUNCTION statement.
- An argument can be an expression that contains variables of simple data types, or simple members of records, or simple elements of arrays.
- You may get unpredictable results if a variable has not yet been assigned a value before it is used as an argument in a CALL statement.

Passing Arguments by Value

How 4GL passes an argument between the calling routine and the function depends on the data type of the argument. Except for variables of data type BYTE or TEXT, arguments are passed to the function *by value*. That is, a copy of the argument is passed. (In this case, changing the value of a formal argument within the function has no effect in the calling routine.)

Passing Arguments by Reference

4GL passes arguments of data type BYTE or TEXT *by reference*. In this case, the function works directly with the actual variable, rather than with a copy. That is, changing a reference to a formal argument in a function changes the corresponding variable in the calling routine. You can use this as a substitute for the RETURNING clause, which does not permit BYTE or TEXT variables. This example show how to pass a BYTE or TEXT argument to a 4GL function:

```
MAIN
    DEFINE resume TEXT
    ...
    LOCATE resume IN MEMORY
    CALL get_resume(resume)
    ...
END MAIN
FUNCTION get_resume(parm)
    DEFINE parm TEXT
    ...
END FUNCTION
```

In this example, the LOCATE statement allocates memory for the TEXT variable, and places a pointer to this variable in **resume**. Any change to **parm** within the **get_resume( )** function also changes the TEXT variable in MAIN.

The RETURNING Clause

The RETURNING clause assigns values returned by the function to variables in the calling routine. To use this feature, you must do the following:

- You need to know how many values the function returns. In the CALL statement, specify that number of variables in the RETURNING clauses.
- If you write the function definition, include expressions in a RETURN statement (page 3-263) to specify values returned by the function.

When returning values to the CALL statement, keep the following in mind:

- The values in the RETURN statement of the FUNCTION must correspond in number and position to the variables specified in the RETURNING clause of the CALL statement. Data types of the RETURNING variables must be compatible (page 3-324) with the RETURN values, but they need not be identical.
- It is an error to specify more variables in the RETURNING clause than the number of values in the RETURN statement of the FUNCTION definition.
- You can return simple or RECORD variables from a function. You cannot, however, return RECORD members of ARRAY, BYTE, or TEXT data types.
- The RETURNING clause passes information by value.

Because they are passed by reference (page 3-18), variables of the BYTE or TEXT data types cannot be included in the RETURNING clause.

Note: *It is an error to specify a RETURNING clause in the CALL statement if the function does not return anything. It is not an error to omit the RETURNING clause when you invoke a function that returns values, if no statement in the calling routine references the returned values.*

Here the returned value in a CHAR variable cannot be longer than 512 bytes. You can use TEXT variables to pass longer character values by reference (page 3-18), rather than using the RETURNING clause.

In the next example, the **get_cust( )** function returns values of **whole_price** and **ret_price** to the CALL statement. 4GL then assigns the **whole_price** and **ret_price** variables to the **wholesale** and **retail** variables in the **price** record:

```
MAIN
    DEFINE price RECORD
        wholesale, retail MONEY
        END RECORD
    ...
    CALL get_cust() RETURNING price.*
    ...
END MAIN

FUNCTION get_cust()
    DEFINE whole_price, ret_price MONEY
    ...
    RETURN whole_price, ret_price
END FUNCTION
```

Restrictions on Returned Character Strings

4GL allocates 5 kilobytes of memory to store character strings returned by functions, in 10 blocks of 512 bytes. This imposes restrictions on the total length of returned character strings that the RETURNING clause can receive, and on the number of long character strings. For example, no returned character value can be longer than 511 bytes (because every string requires a terminating ASCII 0), and no more than 10 of these 511-byte strings can be returned to the calling routine.

Invoking a Function Without CALL

If a function returns a value, you can invoke it without using CALL by simply including it (and any arguments) within an expression in contexts where the returned value is valid; see "Function Calls as Operands" on page 3-332:

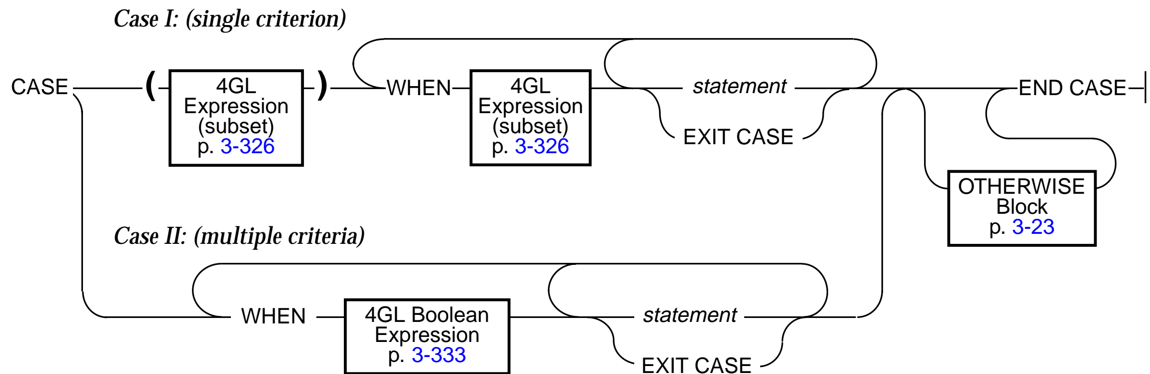
```
IF get_order() THEN
    LET total = total + get_items()
END IF
```

References

DEFINE, FUNCTION, RETURN, WHENEVER

CASE

The CASE statement specifies statement blocks to be executed conditionally, depending on the value of an expression. Unlike IF statements (page 3-124), CASE does not restrict the logical flow of control to only two branches.



statement is an SQL statement or other 4GL statement.

Usage

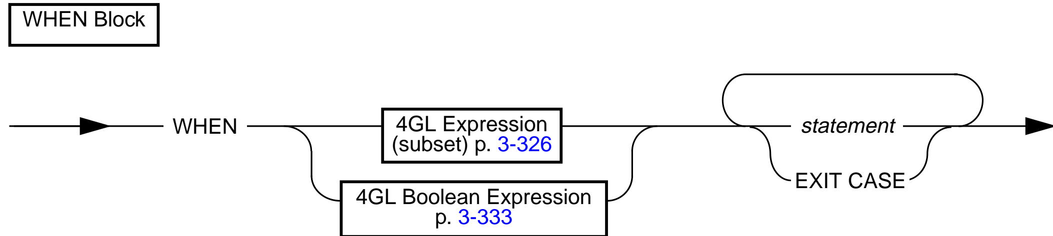
The CASE statement is equivalent to a set of nested IF statements. You can specify two types of CASE statements:

- If you specify an *expression* following the CASE keyword, you must also specify INT, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1) expressions in the WHEN block. (The syntax diagram indicates a *subset* of general 4GL expressions because of these data type restrictions.) 4GL executes the *statement* block if both expressions return the same non-NULL value.
- If no expression follows the CASE keyword, the WHEN block must specify a 4GL *Boolean expression*; if this returns TRUE, then the WHEN block is executed. (See page 3-333, "4GL Boolean Expressions.") This form of CASE typically executes more quickly than the equivalent CASE *expression* form.

There is an implicit EXIT CASE statement at the end of each WHEN block of statements. An implicit or explicit EXIT CASE statement transfers program control to the statement that immediately follows the END CASE keywords.

The WHEN Blocks

Each WHEN block specifies an *expression* and a block of one or more associated *statements*. The WHEN block has the following syntax:



statement is an SQL statement or other 4GL statement.

The data type of the expression that you use in the WHEN blocks depends on whether or not you include an expression following the CASE keyword. If you omit the CASE *expression*, then you must include a *Boolean expression* (one that returns either TRUE or FALSE or NULL) in each of the WHEN blocks. If the Boolean expression is TRUE (non-zero), 4GL executes the corresponding block of statements, as in the following CASE statement:

```
CASE
    WHEN total_price < 1000
    ...
    WHEN total_price = 1000
    ...
    WHEN total_price > 1000
    ...
END CASE
```

If CASE *expression* precedes the first WHEN block, then an INT, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1) expression must follow the WHEN keywords. Each WHEN *expression* must evaluate to a data type compatible with the CASE *expression*. If a WHEN *expression* matches the value of the CASE *expression*, then 4GL executes the statements. In the following example, both **customer_num** and the WHEN *expression* values are of type SMALLINT:

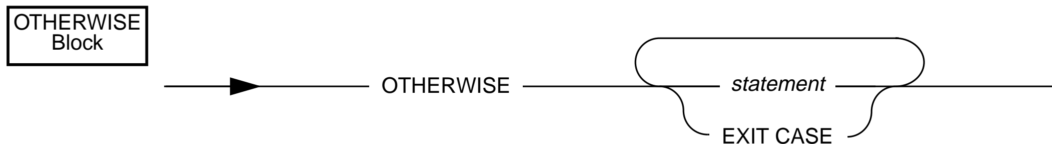
```
CASE (p_customer.customer_num)
    WHEN 101
    ...
    WHEN 102
    ...
END CASE
```

4GL does not execute the *statement* block if the expression in the WHEN block returns FALSE or NULL, or if the CASE *expression* returns NULL. (The IF and WHILE statements and the WHERE clause of a COLOR attribute also treat any NULL value returned from a 4GL Boolean expression as FALSE.)

The OTHERWISE Block

The OTHERWISE keyword specifies statements to be executed when 4GL does not find a matching WHEN block to execute. It has this syntax:

OTHERWISE Block



statement is an SQL statement or other 4GL statement.

4GL executes the OTHERWISE block only if it cannot execute any of the WHEN blocks. If you include the OTHERWISE block, it must follow the last WHEN block. In the next example, if neither 4GL Boolean expression in the WHEN blocks returns TRUE, 4GL invokes the **retry( )** function:

```
WHILE question ...
    CASE
        WHEN answer MATCHES "[Yy]"
            CALL process()
            LET question = FALSE
        WHEN answer MATCHES "[Nn]"
            CALL abort()
        OTHERWISE
            CALL retry()
    END CASE
END WHILE
```

An implied EXIT CASE statement follows the statements in the OTHERWISE block. Unless the OTHERWISE block contains a valid GOTO statement, program control passes to the statement that follows the END CASE statement.

The EXIT CASE Statement

The EXIT CASE statement interrupts processing of the WHEN or OTHERWISE block. When it executes an EXIT CASE statement, 4GL does the following:

- Skips all statements between the EXIT CASE and END CASE keywords.
- Resumes execution at the statement following the END CASE keywords.

Use of the GOTO statement (page 3-122) to leave a WHEN block, rather than an implicit or explicit EXIT CASE statement, may cause run-time error -4518.

The END CASE Keywords

Use the END CASE keywords to indicate the end of the CASE statement construct. These keywords must follow either the last WHEN block or else the OTHERWISE block. After executing the WHEN or OTHERWISE block, 4GL passes control to the statement that follows the END CASE keywords.

In the next example, **quantity** has a SMALLINT value. When **quantity** equals **min_qty**, 4GL executes the statement in the **min_qty** block. When **quantity** equals **max_qty**, 4GL executes the statements in the **max_qty** block.

```
CASE (quantity)
    WHEN min_qty
    ...
    WHEN max_qty
    ...
END CASE
```

In the next example, **print_option** is declared as a global CHAR(1) variable that controls the destination of output from a REPORT program block. The value of **print_option** determines which statement block is executed:

```
CASE (print_option)
    WHEN "f"
        PROMPT " Enter file names for labels >"
            FOR file_name
        IF file_name IS NULL THEN
            LET file_name = "labels.out"
        END IF
        MESSAGE "Printing mailing labels to ",
            file_name CLIPPED," -- Please wait"
        START REPORT labels_report TO file_name
    WHEN "p"
        MESSAGE "Printing mailing labels -- Please wait"
        START REPORT labels_report TO PRINTER
    WHEN "s"
        START REPORT labels_report
        CLEAR SCREEN
END CASE
```

Because WHEN blocks are logically disjunct, exactly one of the START REPORT statements is executed by the CASE statement in this example.

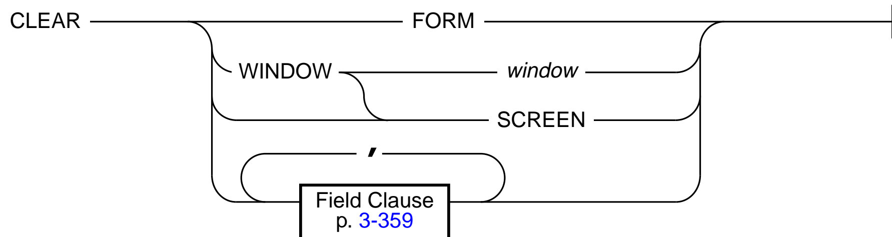
References

FOR, IF, WHILE

CLEAR

The CLEAR statement can clear any of these portions of the screen display:

- The 4GL screen (excluding any open 4GL windows within it).
- Any specified 4GL window.
- All of the fields in the current screen form.
- A list of one or more specified fields in the current screen form.



window is the name of the 4GL window to be cleared.

Usage

The CLEAR statement clears the specified portion of the display. It does *not* change the value of any 4GL variable.

The CLEAR FORM Option

Use CLEAR FORM to clear all fields of the form in the current 4GL window:

```
CLEAR FORM
```

The CLEAR FORM option has no effect on other parts of the screen display.

The CLEAR WINDOW Option

Use CLEAR WINDOW *window* to clear a specified 4GL window, for *window* a 4GL identifier that was declared in an OPEN WINDOW statement:

```
CLEAR WINDOW threshold
```

If the *window* that you specify has a border, the CLEAR WINDOW statement does not erase the border. You can specify any 4GL window, including one that is not the current window, but the CLEAR WINDOW statement does not affect which 4GL window is the current 4GL window in the window stack.

The CLEAR WINDOW SCREEN Option

If you specify CLEAR WINDOW SCREEN, then 4GL does the following:

- Clears the 4GL screen, except for the area occupied by any open 4GL windows.
- Leaves any information in the open 4GL windows untouched.
- Does not change the current 4GL window setting.

As in several other 4GL statements, the keyword SCREEN here specifies the 4GL screen.

The CLEAR SCREEN Option

Use the CLEAR SCREEN option to make the 4GL screen the current 4GL window, and to clear everything on it, including the Prompt, Message, and Error lines. In the next example, choosing the Exit option clears the screen and terminates the MENU statement:

```
MENU "ORDERS"
    COMMAND "Add-order"
            "Enter new order into database and print invoice"
            HELP 301
        CALL add_order( )
        ...
    COMMAND "Exit"
            "Return to MAIN MENU"
            HELP 305
        CLEAR SCREEN
        EXIT MENU
END MENU
```

The CLEAR *field* Option

Use the CLEAR *field* option to clear the specified field or fields in a form that the current 4GL window displays. For the syntax of the *field* clause, see the section "Field Clause" (page 3-359) later in this chapter. The next example clears the fields named **fname**, **lname**, **address1**, **city**, **state**, and **zipcode**:

```
CLEAR fname, lname, address1, city, state, zipcode
```

If you specify *table*.* (for *table* a name or alias from the TABLE section of the form specification file), CLEAR clears all the fields associated with columns of that table. (See "INSTRUCTIONS Section" in Chapter 5 for a description of screen records and screen arrays that the *record*.* notation can reference.)

For example, the following program fragment clears the **orders** screen record and the first four records of the **s_items** screen array:

```
FOREACH order_list INTO p_orders.*
    CLEAR s_orders
    FOR idx = 1 TO 4
        CLEAR s_items[idx].*
    END FOR
    DISPLAY p_orders.* TO orders.*
    ...
END FOREACH
```

If a screen form is in the current 4GL window, then the following statement clears all the screen fields that are not associated with database columns:

```
CLEAR FORMONLY.*
```

Any fields that you associated with database columns in the ATTRIBUTES section of the form specification file are *not* affected by this statement.

References

CLOSE FORM, CLOSE WINDOW, CURRENT WINDOW, DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, OPEN FORM, OPEN WINDOW, OPTIONS

CLOSE FORM

The CLOSE FORM statement releases the memory required for a form.

```
CLOSE FORM   form ─────────────────────────────────────────┤
```

form   is the name of the 4GL screen form to be cleared from memory.

Usage

When it executes the OPEN FORM statement, 4GL loads the compiled screen form into memory. Until you close the form, it remains in memory. To regain the memory allocated to a form, you can use the CLOSE FORM statement. For example, the following program fragment opens and displays the **o_cust** form, and then closes both the form and the 4GL window **cust_w**:

```
OPEN WINDOW cust_w AT 3,5 WITH 19 ROWS, 72 COLUMNS
OPEN FORM o_cust FROM "custform"
DISPLAY FORM o_cust ATTRIBUTE(MAGENTA)
...
CLOSE FORM o_cust
CLOSE WINDOW cust_w
```

If you open the form by using the WITH FORM option of the OPEN WINDOW statement, you do not need to use the CLOSE FORM statement before closing the 4GL window. In this case, CLOSE WINDOW both closes the form (releasing the memory allocated to that form) and closes the 4GL window.

CLOSE FORM affects memory use only, not the logic of the 4GL program. After you use CLOSE FORM to release the memory that supports a form, its name is no longer associated with the form. If you subsequently try to redisplay the form, an error message results. If you execute a new OPEN FORM or OPEN WINDOW statement that specifies the same *form* name that an OPEN FORM or OPEN WINDOW statement referenced previously, 4GL automatically closes the previously opened form before opening the new form.

References

CLOSE WINDOW, DISPLAY FORM, OPEN FORM, OPEN WINDOW

CLOSE WINDOW

The CLOSE WINDOW statement closes a specified 4GL window.

```
CLOSE WINDOW   window ──────────────────────────────────┤
```

window is the identifier of the 4GL window to be closed.

Usage

The CLOSE WINDOW statement causes 4GL to take the following actions:

- Clears the specified 4GL window from the 4GL screen, and restores any underlying display.
- Frees all resources used by the 4GL window, and deletes it from the **4GL** window stack.
- If the OPEN WINDOW statement included the WITH FORM clause, the CLOSE WINDOW statement closes both the form and the **4GL** window.

4GL maintains an ordered list of open 4GL windows, called the *window stack*. When you open a new 4GL window, the new window is added to the stack and becomes the *current window*, meaning that it occupies the top of the stack. If you close the current window, the next 4GL window on the stack becomes the new current window. If you close any other window, 4GL deletes it from the window stack, leaving the current window unchanged.

Closing a window has no effect on values of variables that were set while the window was open.

This program fragment opens and closes a 4GL window called **stock_w**:

```
OPEN WINDOW stock_w AT 7, 3 WITH 6 ROWS, 70 COLUMNS
...
CLOSE WINDOW stock_w
```

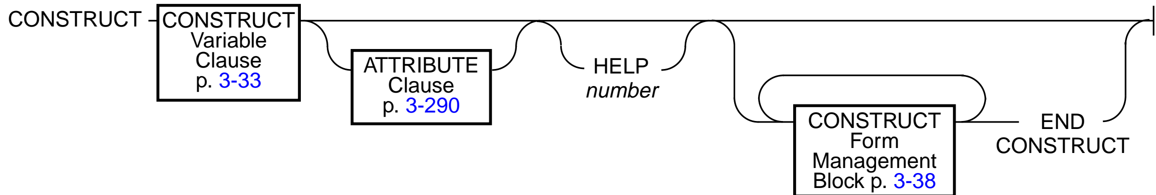
You *cannot* specify CLOSE WINDOW SCREEN. If *window* is currently being used for input, CLOSE WINDOW generates a run-time error. For example, you cannot close the current 4GL window while a CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or MENU statement is executing.

References

CLEAR, CLOSE FORM, CURRENT WINDOW, OPEN WINDOW, OPTIONS

CONSTRUCT

The CONSTRUCT statement stores in a character variable the 4GL Boolean expression that corresponds to the *query by example* that a user specifies. You can use this character variable as the WHERE clause of a SELECT statement.



number is a literal integer (page 3-340) to specify a Help message number.

Usage

The CONSTRUCT statement is among the statements required to allow users to perform a *query by example*. Query by example lets a user query a database by specifying values or ranges of values for screen fields that correspond to database columns. 4GL converts these values into a Boolean expression that specifies search criteria in the WHERE clause of a SELECT statement.

The CONSTRUCT statement can also control the environment in which the user enters search criteria, and can restrict the values that the user enters. To use the CONSTRUCT statement, you must do the following:

- Define *fields* linked to database columns in a form specification file.
- Declare a *character variable* with the DEFINE statement.
- Open and display the *screen form* with either of the following:
  - OPEN FORM and DISPLAY FORM statements.
  - OPEN WINDOW statement with a WITH FORM clause.
- Use the CONSTRUCT statement to store in the character variable a *Boolean expression* based on search criteria that the user enters in the fields.

The CONSTRUCT statement activates the *current form*. This is the form most recently displayed or, if you are using more than one 4GL window, the form currently displayed in the current window. You can specify the current window by using the CURRENT WINDOW statement. When the CONSTRUCT statement completes execution, the form is deactivated.

When it encounters the CONSTRUCT statement, 4GL does the following:

1. Displays blank spaces in all the screen fields of the CONSTRUCT field list.
2. Moves the screen cursor to the first screen field in that list.
3. Waits for the user to enter some value as search criteria in the field. (For fields where the user enters no value, *any* value in the corresponding database column satisfies the search criteria.)

After the user chooses Accept, CONSTRUCT uses AND operators to combine field values as search criteria in a Boolean expression, and stores this in a character variable. By performing the following steps, you can use this variable in a WHERE clause to search the database for matching rows:

1. Concatenate the character variable that contains the Boolean expression with one or more character strings to create the string representation of an SQL statement to be executed. (The Boolean expression generated by the CONSTRUCT statement is typically used to create SELECT statements.)
2. Use the PREPARE statement to create an executable SQL statement from the character string that was generated in the previous step.
3. Execute the prepared statement in one of the following ways:
   - Use an SQL cursor with DECLARE and FOREACH statements (or else OPEN and FETCH statements) to execute a prepared SELECT statement that includes no INTO clause.
   - Use the EXECUTE statement to execute other SQL statements.

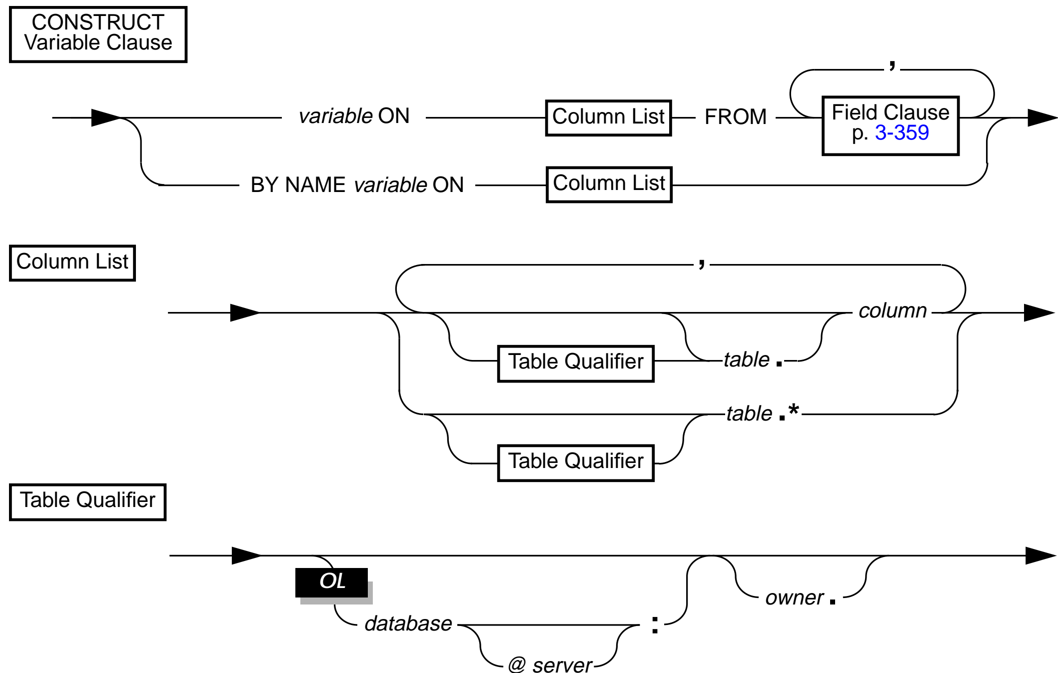
Environment variables that format data values, such as DBDATE, DBTIME, DBFORMAT, DBFLTMASK, or DBMONEY, have no effect on the contents of the Boolean expression.

The following are among the topics that are described in this section:

Topic	Page
The CONSTRUCT Variable Clause	3-33
The ATTRIBUTE Clause	3-37
The HELP Clause	3-37
The CONSTRUCT Form Management Blocks	3-38
The END CONSTRUCT Keywords	3-47
Using Built-In Functions and Operators	3-47
Query by Example	3-48
Positioning the Screen Cursor	3-50
Editing During a CONSTRUCT Statement	3-52
Completing a Query	3-52

The CONSTRUCT Variable Clause

The CONSTRUCT variable clause specifies a character variable to store search criteria, and one or more screen fields in which the user can enter search criteria for database columns. The CONSTRUCT variable clause has this syntax:



column	is the name of some database column in *table.*
database	is the name of the database in which the *table* resides.
owner	is the username of the owner of the table containing the *column.*
server	is the name of the host system where *database* resides.
table	is the name or synonym of a table or view in the specified *database*, in DBPATH, or in the current database (page 3-60) most recently, specified by a DATABASE statement within this program block.
variable	is the name of a CHAR or VARCHAR variable that stores the 4GL Boolean expression that the CONSTRUCT statement creates, summarizing the user-entered search criteria.

This clause temporarily binds the specified screen fields to database columns. It allows you to identify database columns for which the user can enter search criteria. You can map the fields implicitly (with the BY NAME clause) or explicitly (with the FROM clause). With either method, each field and

corresponding column must be the same or compatible data types. The order of fields in the FROM clause determines the default sequence in which the screen cursor moves from field to field on the screen form. Within a screen array, you can specify only one screen record.

4GL "constructs" a character variable by associating each *column* name in the ON clause with *search criteria* that the user enters into the corresponding field (as specified in the FROM clause, or implied by the BY NAME keywords). You can use the information stored in character variable in the WHERE clause of a prepared SELECT statement to retrieve rows from the database. To avoid overflow, the length of the variable should be a several times the total length of all the fields, since the Boolean expression includes additional operators.

The BY NAME Clause

You can use the BY NAME clause when the fields on the screen form have the same names as the corresponding columns in the ON clause. The BY NAME clause maps the form fields to columns implicitly. The user can query only the screen fields implied in the BY NAME clause. The following CONSTRUCT statement, for example, assigns search criteria to the variable **query_1**:

```
CONSTRUCT BY NAME query_1 ON company, address1, address2,
    city, state, zipcode
```

The user can enter search criteria in the fields named **company**, **address1**, **address2**, **city**, **state**, and **zipcode**. Because these fields have the same names as the columns specified after the ON keyword, the statement uses the BY NAME clause. If the field names do not match the column names, you must use the FROM clause instead of the BY NAME clause. This functionally equivalent CONSTRUCT statement uses the FROM clause:

```
CONSTRUCT query_1
    ON company, address1, address2, city, state, zipcode
    FROM company, address1, address2, city, state, zipcode
```

If the column names in a CONSTRUCT BY NAME statement are associated with field names in a screen array, the construct takes place in the *first row* of the screen array. If you want the CONSTRUCT to take place in a different row of the screen array, you must use the FROM clause, not the BY NAME clause.

You cannot preface column names with a qualifier that includes an *owner* name, a *server* name, or a *pathname* when you use the BY NAME clause. Use the FROM clause to specify *table aliases* in the field list when the qualifier of any column name requires an owner name, a server name, or a pathname.

The ON Clause

The ON clause specifies a list of database columns for which the user will enter search criteria. These columns do not have to be from the same table. If the CONSTRUCT statement includes the BY NAME keywords, be sure that the fields on the screen form have the same names as the columns listed after the ON keyword. If the CONSTRUCT statement includes a FROM clause, the expanded list of columns in the ON clause must correspond in order and in number to the expanded list of fields in the FROM clause.

You can use the notation *table.** (page 3-363), meaning "every column in *table,*" for all or part of the column list. The order of columns within *table* depends on their order in the **syscolumns** system catalog table when you compile your program. If the ALTER TABLE statement has changed the order, the names, the data types, or the number of the columns in *table* since you compiled your program, then you might need to modify your program and its screen forms that reference that table.

The following example uses the **customer.*** notation as a macro for listing all columns in the **customer** table and **cust.*** as a macro for all the fields in the **customer** screen record:

```
CONSTRUCT query_1 ON customer.* FROM cust.*
```

The FROM Clause

The FROM clause specifies a list of screen fields or screen records in the form. You cannot use the FROM clause if you include the BY NAME clause, but you *must* use the FROM clause if any of the following conditions are true:

- The names of fields on the screen form are different from the names of the corresponding database columns in the ON clause.
- You want to reference fields in a screen array beyond the first record.
- You specify additional qualifiers for *table.column* in the ON clause (for example, for external or non-unique *table* names, or to reference the *owner* of a table if the database is ANSI-compliant).

The user can position the cursor only in fields specified in the FROM clause. The list of fields in the FROM clause must correspond in order and in number to the list of database *columns* in the ON clause, as in this example:

```
CONSTRUCT query_1 ON stock_num, manu_code, description
    FROM stock_no, m_code, descr
```

If you use the *record.** notation for all or part of a field list, be sure that the implied order of fields corresponds to the order of columns in the ON clause. (The order of fields in *record* depends on its declaration in the form.)

In the following CONSTRUCT statement, the field list includes the **stock_num** and **manu_code** fields, as well as the screen record **s_stock.*** that corresponds to the remaining columns in the **stock** table:

```
CONSTRUCT query_1 ON stock.*
    FROM stock_num, manu_code, s_stock.*
```

To specify an individual field of a screen array in the FROM clause, use the notation *record* **[** *n* **]** **.** *field name* to indicate the row in which the CONSTRUCT takes place. For example, the following CONSTRUCT statement allows the user to enter search criteria in the third record of screen array **s_items**:

```
CONSTRUCT query_1 ON items.* FROM s_items[3].*
```

The FROM clause is required when the field list includes an alias representing a table, view, or synonym name that includes any qualifier. For example, in the following CONSTRUCT statement, **cust** is a table alias declared in the form specification file for the **actg.customer** table, where **actg** is an *owner* prefix. This table alias must be prefixed to each field name in the FROM clause, because the column qualifiers in the ON clause include an owner name:

```
CONSTRUCT query_1 ON
    actg.customer.fname, actg.customer.lname,
    actg.customer.company
    FROM cust.fname, cust.lname, cust.company
```

The ATTRIBUTE Clause

For information about the ATTRIBUTE clause, see page 3-290. This section describes how to use the ATTRIBUTE clause in a CONSTRUCT statement. You can use the ATTRIBUTE clause to apply display attributes to the fields specified implicitly in the BY NAME clause or explicitly in the FROM clause. If you use the ATTRIBUTE clause, the following attributes do not apply to the fields:

- Default attributes listed in **syscolatt** table. (See the description of "The upscol Utility" on page B-5.)
- Default attributes (REVERSE and COLOR) in the form specification file.
- The NOENTRY attribute in the form specification file.

The CONSTRUCT attributes temporarily override any display attributes set by the INPUT ATTRIBUTE clause of an OPTIONS or OPEN WINDOW statement. Attributes in the ATTRIBUTE clause of CONSTRUCT apply to all the fields in the field list, but only during the current activation of the form. When the user deactivates the form, the form reverts to its previous attributes.

The following CONSTRUCT statement includes an ATTRIBUTE clause that specifies CYAN and REVERSE for values entered in screen fields that have the same names as the columns in the **customer** table:

```
CONSTRUCT BY NAME query_1 ON customer.*
        ATTRIBUTE (CYAN, REVERSE)
```

The HELP Clause

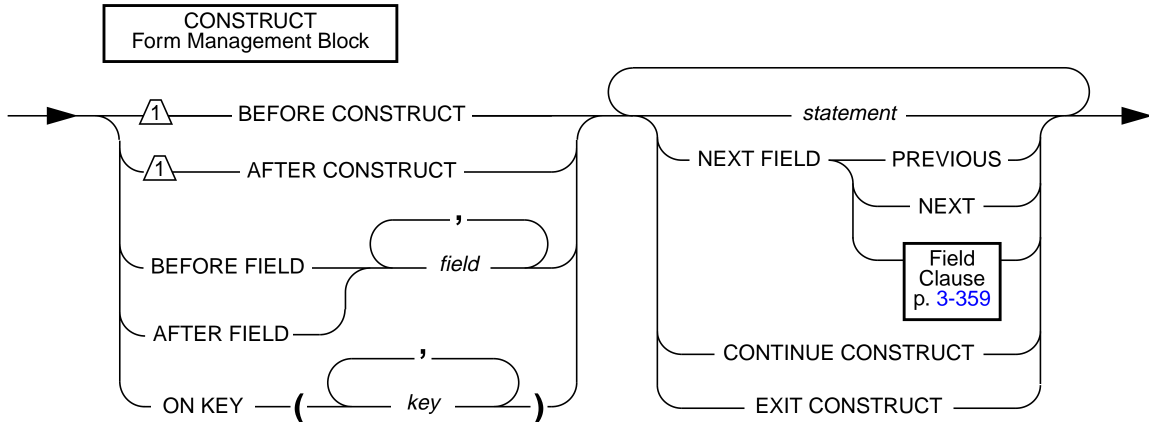
The HELP clause specifies the number of the Help message associated with the CONSTRUCT statement. This message appears in the Help window (page 2-22) when the user presses the Help key from any field in the field list. The Help key is CONTROL-W by default, but the OPTIONS statement (page 3-228) can assign a different physical key as the Help key.

You create Help messages in an ASCII file. For details of how to create a run-time version of the Help file, see "The mkmessage Utility" on page B-2. An error occurs if 4GL cannot open the Help file, or if *number* is not in the Help file, or if the specified value is greater than 32,767. To provide field-level Help, use an ON KEY clause with the INFIELD( ) operator and SHOWHELP( ) function; both are described in Chapter 4.

If you provide messages to assist the user through an ON KEY clause, rather than by the HELP clause, the message must be displayed in a 4GL window within the 4GL screen, rather than in the separate Help window.

The CONSTRUCT Form Management Blocks

Each CONSTRUCT form management block includes a *statement block* of at least one statement, and an *activation clause* that specifies when to execute the *statement block*. The *activation clause* and *statement block* correspond respectively to the left-hand and right-hand syntax elements in this diagram:



field is the name of a field that was either explicitly or implicitly referenced in the CONSTRUCT Variable clause (page 3-33).

key is one of the keywords that are listed in the section "The ON KEY Blocks" on page 3-41.

statement is an SQL or other 4GL statement.

You can use a CONSTRUCT form management block to specify:

- Statements to execute before and after the query by example.
- Statements to execute before and after a given field.
- Statements to execute if a user presses some key sequence.
- The next field to which to move the screen cursor.
- When to exit from the CONSTRUCT statement.

4GL executes the *statements* in the block according to the following events:

- The *fields* into which and from which the user moves the cursor.
- The *keys* that the user presses.

Statements can include CONTINUE CONSTRUCT and EXIT CONSTRUCT, the NEXT FIELD clause, and most 4GL and SQL statements. The CONSTRUCT, or INPUT statement is not valid in a CONSTRUCT form management block, but you can call a function that executes CONSTRUCT, INPUT, or INPUT ARRAY statements in a different **4GL** window.

4GL temporarily deactivates the form while executing statements in a form management block. After executing the statements, 4GL reactivates the form, allowing the user to continue modifying values in fields.

The Precedence of Form Management Blocks

The CONSTRUCT statement can list form management blocks in any order. You should develop some consistent ordering, however, so that your code is more readable than if the blocks were randomly ordered. When you use one or more form management blocks, you must include the END CONSTRUCT statement to terminate the CONSTRUCT statement. If you include several form management blocks, 4GL processes them in the following sequence, regardless of the order in which they appear in the CONSTRUCT statement:

1. BEFORE CONSTRUCT (executed before the user begins entering values)
2. BEFORE FIELD (executed before the user enters values in a specified field)
3. ON KEY (executed after the user presses a specified key)
4. AFTER FIELD (executed after the user enters values in a specified field)
5. AFTER CONSTRUCT (executed after the user has finished entering values)

If you include no form management blocks, the program waits while the user enters values in the fields. When the user accepts the values in the form, the CONSTRUCT statement terminates, and control passes to the next statement.

The BEFORE CONSTRUCT Block

4GL sets the values of all the screen fields to blank spaces when it first executes a CONSTRUCT statement. You can use the BEFORE CONSTRUCT block to supply different initial default values for the fields.

CONSTRUCT executes the statements in the BEFORE CONSTRUCT block once before it allows the user to perform the query by example. You can use DISPLAY statements in the BEFORE CONSTRUCT block to populate the fields; DISPLAY initializes the field buffers to the displayed values. To determine the first field where criteria can be entered, you can use the NEXT FIELD clause.

For example, the following DISPLAY statement assigns the values in the **rec** program record to the field buffers associated with the **srec** screen record:

```
BEFORE CONSTRUCT
    DISPLAY rec.* TO srec.*
    NEXT FIELD lname
```

The CONSTRUCT statement can include only one BEFORE CONSTRUCT block. You cannot include the FIELD_TOUCHED( ) operator in this block.

The BEFORE FIELD Blocks

This block specifies statements that are associated with a specific screen field. 4GL executes the BEFORE FIELD block statements of a field whenever the screen cursor enters the field, and before the user types search criteria.

You can use a NEXT FIELD clause within a BEFORE FIELD block to restrict access to a field. You can also use a DISPLAY statement within a BEFORE FIELD block to display a default value in a field.

The following program fragment defines two BEFORE FIELD blocks. The first block uses the NEXT FIELD clause to limit access to the **salary** field to certain users. The second block displays the current date in the **q_date** field:

```
CONSTRUCT BY NAME query_1 ON employee.*
    BEFORE FIELD salary
        IF (username <> "manager") AND (username <> "admin")
            THEN NEXT FIELD NEXT
        END IF
    BEFORE FIELD q_date
        LET query_date = TODAY
        DISPLAY query_date TO q_date
END CONSTRUCT
```

The ON KEY Blocks

The *statements* in the appropriate ON KEY block are executed if the user presses the activation key corresponding to one of your key specifications. These are the keywords that you can specify for *key*:

ACCEPT	HELP	NEXT *or*	RETURN
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS *or*	TAB
ESC *or* ESCAPE	LEFT	PREVPAGE	UP

F1 *through* F64

CONTROL-*char* (except A, D, H, I, J, L, M, R, or X)

Like other keywords of **4GL**, you can specify these in uppercase or lowercase.

Some keys need special consideration if you assign them in an ON KEY block:

Key	Special Considerations
ESC or ESCAPE	You must use the OPTIONS statement to specify another key as the Accept key, because ESCAPE is the default Accept key.
INTERRUPT	You must execute a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, **4GL** executes the ON KEY block and sets **int_flag** to non-zero, but does not terminate the CONSTRUCT statement. Similarly, **4GL** executes the statements in the ON KEY block and sets **quit_flag** to non-zero if the DEFER QUIT statement has been executed when the user presses the Quit key.
CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these keys for field editing.
I, J, and M	The standard meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is lost to the user. Instead, the key is trapped by **4GL** and activates the commands in the ON KEY block. For example, if CONTROL-M appears in an ON KEY block, the user cannot press RETURN to advance the cursor to the next field. If you specify one of these keys in an ON KEY block, be careful to restrict the scope of the statement.

You may not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

The following ON KEY block calls a function that display a Help message. Here the BEFORE CONSTRUCT block informs the user how to access Help:

```
BEFORE CONSTRUCT
    MESSAGE "Press F8 or CTRL-Y for Help"
ON KEY (f8, control-Y)
    CALL customer_help()
```

If an ON KEY block is activated during data entry, 4GL does the following:

1. Suspends the input of the current field.
2. Preserves the input buffer that contains the characters the user has typed.
3. Executes the statements in the corresponding ON KEY clause.
4. Restores the input buffer for the current screen field.
5. Resumes input in the same field, with the cursor at the end of the buffered list of characters.

You can change this default behavior by including statements to perform the following tasks in the ON KEY block:

- Resuming input in another field by using the NEXT FIELD keywords.
- Changing the input buffer value for the current field by assigning a new value to the corresponding variable and then displaying the value.

The AFTER FIELD Blocks

4GL executes the AFTER FIELD block associated with a field when the cursor leaves the field. This includes instances when the user chooses Accept. When the NEXT FIELD clause appears in an AFTER FIELD block, 4GL places the cursor in the specified field and ignores the Accept keystroke. If an AFTER FIELD block exists for each field, and if a NEXT FIELD clause appears in every AFTER FIELD block, then the user is unable to leave the form.

The following program fragment checks for the Accept key, and terminates execution of CONSTRUCT if the Accept key was chosen:

```
AFTER FIELD status
    IF NOT GET_LASTKEY( ) = ACCEPT_KEY THEN
        LET p_stat = GET_FLDBUF(status)
        IF p_stat MATCHES "married" THEN
            NEXT FIELD spouse_name
        END IF
    END IF
END CONSTRUCT
```

As noted in the section "Completing a Query" on page 3-52, the user can terminate the CONSTRUCT statement by choosing Accept, Interrupt, or Quit, or by pressing the TAB or RETURN key after the last form field. You can use the AFTER FIELD clause with the NEXT FIELD keywords on the last field to override this default termination. (Alternatively, you can specify INPUT WRAP in an OPTIONS statement to achieve the same effect.)

The AFTER CONSTRUCT Block

4GL executes the statements in the AFTER CONSTRUCT block after the user chooses Accept and before 4GL constructs the string containing the Boolean expression. You can use the AFTER CONSTRUCT block to validate, save, or alter the values of the screen field buffers. The section titled "Using Built-In Functions and Operators" on page 3-47 describes built-in functions and operators of 4GL that commonly appear in the AFTER CONSTRUCT block.

You can specify CONTINUE CONSTRUCT or NEXT FIELD in this block to return the cursor to the form. If you include these in the AFTER CONSTRUCT block, be sure that it appears within a conditional statement. Otherwise, the user cannot exit from the CONSTRUCT statement and leave the form.

In the following program fragment, a CONTINUE CONSTRUCT statement appears in an IF statement. If the user does not specify any selection criteria, then 4GL returns the screen cursor to the form:

```
AFTER CONSTRUCT
    IF NOT FIELD_TOUCHED(orders.*) THEN
        MESSAGE "You must indicate at least one ",
            "selection criteria."
        CONTINUE CONSTRUCT
    END IF
```

See also the section "Searching for All Rows" on page 3-50.

4GL executes the statements in the AFTER CONSTRUCT block when the user presses any of the following keys:

- The Accept key.
- The Interrupt key (if DEFER INTERRUPT has executed).
- The Quit key (if the DEFER QUIT statement has executed).

The AFTER CONSTRUCT block is *not* executed in the following situations:

- The user presses the Interrupt or Quit key and the DEFER INTERRUPT or DEFER QUIT statement, respectively, has not been executed. In either case, the program terminates immediately, and no query is performed.
- The EXIT CONSTRUCT statement terminates the CONSTRUCT statement.

A CONSTRUCT statement can include only one AFTER CONSTRUCT clause.

The NEXT FIELD Clause

While the CONSTRUCT statement is executing, 4GL moves the screen cursor from field to field in the order specified in the FROM clause, or in the order implied by the ON clause of the CONSTRUCT BY NAME statement). You can use the NEXT FIELD clause, however, to control cursor movement.

You can specify any of the following fields in the NEXT FIELD clause:

- The *next* field, as defined by the order of fields in the CONSTRUCT statement. In this case, specify the NEXT keyword.
- The *previous* field, as defined by the order of fields in the CONSTRUCT statement. In this case, specify the PREVIOUS keyword.
- Any other field in the current form. In this case, specify the name of the field (from the ATTRIBUTES section of the form specification file).

The NEXT FIELD keywords can appear in a BEFORE CONSTRUCT block (for example, to position the cursor at a different starting field) and in a BEFORE FIELD block (for example, to restrict access to a field), but they are more commonly used in AFTER FIELD, ON KEY, or AFTER CONSTRUCT blocks.

Use NEXT FIELD only if you want the cursor to deviate from the default field order. 4GL immediately positions the cursor in the form when it encounters the NEXT FIELD clause, without executing any statements that immediately follow the NEXT FIELD clause in the same statement block. The following -

program fragment demonstrates this. Here the **qty_help( )** function cannot be invoked, because its CALL statement is positioned after the NEXT FIELD clause:

```
ON KEY (CONTROL_B, F4)
    IF INFIELD(stock_num) OR INFIELD(manufact) THEN
        CALL stock_help( )
        NEXT FIELD quantity
        CALL qty_help( ) -- function is never called
    END IF
```

The following program segment includes NEXT FIELD clauses in ON KEY and AFTER FIELD blocks. The user triggers the ON KEY block by pressing CONTROL-B or F4. If the cursor is in the **stock_num** field or **manufact** field, 4GL calls the **stock_help( )** function. When 4GL returns from the **stock_help( )** function, the NEXT FIELD clause moves the cursor to the **quantity** field.

The user executes the AFTER FIELD block by moving the cursor out of the **zipcode** field. The FIELD_TOUCHED( ) operator checks whether the user entered a value into the field. If this returns TRUE, then GET_FLDBUF( ) retrieves the value entered into the field during a query, and assigns it to the **p_zipcode** variable. If the first character in the **p_zipcode** variable is not a 5, 4GL displays an error, clears the field, and returns the cursor to the field.

```
ON KEY (CONTROL_B, F4)
    IF INFIELD stock_num) OR INFIELD(manufact) THEN
        CALL stock_help( )
        NEXT FIELD quantity
    END IF

AFTER FIELD zipcode
    IF FIELD_TOUCHED(zipcode) THEN
        LET p_zipcode = GET_FLDBUF(zipcode)
        IF p_zipcode[1,1] <> "5" THEN
            ERROR "You can only search area 5."
            CLEAR zipcode
            NEXT FIELD zipcode
        END IF
    END IF
```

Do not use NEXT FIELD clauses to move the cursor across every field in a form. If you want the cursor to move in a specific order, list the fields in the CONSTRUCT statement in the desired order. In most situations, NEXT FIELD

appears in a conditional statement. The NEXT FIELD clause *must* appear in a conditional statement when it appears in an AFTER CONSTRUCT block; otherwise, the user cannot exit from the query.

The CONTINUE CONSTRUCT Statement

The CONTINUE CONSTRUCT statement skips all subsequent statements in the CONSTRUCT statement, and returns the cursor to the screen form at the last field occupied. This statement is useful where program control is nested within multiple conditional statements and you want to return control to the user. It is also useful in an AFTER CONSTRUCT block, where you can examine field buffers and, depending on their contents, return the cursor to the form.

For example, in the following program fragment, an IF statement tests for an entered value in any field.

```
CONSTRUCT BY NAME query1 ON customer.*
...
    AFTER CONSTRUCT
        IF NOT FIELD_TOUCHED(customer.*) THEN
            PROMPT "Do you really want to see ",
                "all customer rows? (y/n)" FOR CHAR answer
            IF answer MATCHES "[Nn]" THEN
                CONTINUE CONSTRUCT
            END IF
        END IF
END CONSTRUCT
```

If none was entered, the user is prompted to indicate whether to retrieve all customer records. If the user types `N` or `n`, then 4GL executes the CONTINUE CONSTRUCT statement and positions the cursor in the form, giving the user another chance to enter selection criteria in the last field occupied. If the user types any other key, the IF statement terminates, and control passes to the END CONSTRUCT statement.

Note: *Compare this method of detecting and handling the absence of search criteria to the examples in the sections "The AFTER CONSTRUCT Block" (page 3-43) and "Searching for All Rows" (page 3-50).*

When a test in an AFTER CONSTRUCT clause identifies a field that requires action by the user, specify NEXT FIELD, rather than CONTINUE CONSTRUCT, to position the cursor in the field.

The EXIT CONSTRUCT Statement

The EXIT CONSTRUCT statement causes 4GL to do the following:

- Skip all statements between EXIT CONSTRUCT and END CONSTRUCT.
- Terminate the process of constructing the query by example.
- Create the Boolean expression and store it in the character variable.
- Resume execution at the statement after the END CONSTRUCT keywords.

If it encounters the EXIT CONSTRUCT statement, 4GL does not execute the statements in the AFTER CONSTRUCT block, if that block is present.

The END CONSTRUCT Keywords

The END CONSTRUCT keywords indicate the end of the CONSTRUCT statement construct. These keywords should follow the last CONSTRUCT form management block. The END CONSTRUCT keywords are required only if you specify one or more CONSTRUCT form management blocks.

Using Built-In Functions and Operators

The CONSTRUCT statement supports built-in functions and operators of 4GL. (For more about the built-in 4GL functions and operators, see Chapter 4.) The following features allow you to access field buffers and keystroke buffers:

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has "touched" (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or Arrow keys) does not mark a field as touched. This operator also ignores the effect of statements that appear in the BEFORE CONSTRUCT control block. For example, you can assign values to fields in the BEFORE CONSTRUCT control block without having the fields marked as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form.
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user while in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is specified as its operand is the name of the current field.

Each field in a form has only one field buffer, and a buffer cannot be used by two statements simultaneously. If a CONSTRUCT statement calls a function that includes an INPUT or CONSTRUCT statement, and both statements use

the same form, you may overwrite one or more of the field buffers. Fields used in a CONSTRUCT statement should not be reused in an INPUT or another CONSTRUCT statement until the first CONSTRUCT statement finishes.

If you plan to display the same form more than one time and will access the form fields, you should open a new window and open and display a second copy of the form. 4GL allocates a separate set of buffers to each form, and you can be certain that your program is retrieving the correct field values.

Query by Example

The CONSTRUCT statement allows users of your application to specify *search criteria* for retrieving rows from the database. The user does this by entering values (or ranges of values) into the fields of the screen form. This process is called a *query by example*. The user can use symbols to search for data values less than, equal to, greater than, or within a range.

The CONSTRUCT statement supports the following symbols:

Symbol	Meaning	Data Type Domain	Pattern
=	equal to	all simple SQL types	=x, =
>	greater than	all simple SQL types	>x
<	less than	all simple SQL types	<x
>=	greater than or equal to	all simple SQL types	>=x
<=	less than or equal to	all simple SQL types	<=x
<>	not equal to	all simple SQL types	<>x
:	range	all simple SQL types	x:y
..	range	DATETIME, INTERVAL	x..y
*****	wildcard for any string	CHAR, VARCHAR	*x, x*, *x*
?	single-character wildcard	CHAR, VARCHAR	?x, x?, ?x?, x??
|	logical OR	all simple SQL types	a | b...
[ ]	list of values (see next page)	CHAR, VARCHAR	[xy]*, [xy]?

The user cannot perform a query-by-example on BYTE or TEXT fields, nor on FORMONLY fields. The following list explains the symbols in the table above.

x The **x** means any *value* appropriate to the data type of the field. The *value* must immediately follow any of the first six symbols in the preceding table. Do not leave a space between a symbol and a *value*.

=x The equal sign (**=**) is the default symbol for non-character fields, and for character fields in which the search value contains no wildcards. If the user enters a character value that does not contain a wildcard character, CONSTRUCT produces the Boolean expression:

char-column = "*value*"

= The equal sign (**=**) with no *value* searches for a NULL value. The user must explicitly enter the equal sign to find any character value that is also used as a search criteria symbol.

>, <, >=, <=, <> These symbols imply an ordering of the data. For character fields, "*greater than*" means *later* in the ASCII sequence (where a > A > 1), as listed in Appendix A. For DATE and DATETIME data "*greater than*" means *after*. For INTERVAL data, it means a *longer* span of time.

A query by example cannot combine these relational operators with the range, wildcard, or logical operators that are described later. Any characters that follow a relational operator are interpreted as literals.

Besides the relational operators, the user can specify a range, or use syntax like that of the MATCHES operator to search for patterns in character values.

: The colon in **x: y** searches for all values between the **x** and **y** value, inclusive. The **y** value must be larger than the **x** value. The search criterion 1: 10 would find all rows with a value in that column from 1 through 10. For character data, the range specifies values in the ASCII collating sequence between **x** and **y**. (For DATETIME and INTERVAL fields, use instead the .. symbol to specify ranges.)

.. Substitute two periods (..) for the colon in DATETIME and INTERVAL ranges to avoid ambiguity with field separators in *hh*:*mm*:*ss* values.

* The asterisk (`*`) is a character string wildcard, representing zero or more characters. Use the asterisk character as follows:

- The search value `*ts*` in a field specifies all strings containing the letters **ts**, such as the strings "`Watson`" and "`Albertson`."
- The search value `S*` specifies all strings beginning with the letter **s**, including the strings "`S`," "`Sadler`," and "`Sipes`."
- The search value `*er` specifies all strings that end in the letters **er**, such as the strings "`Sadler`" and "`Miller`."

? The question mark (`?`) is the single-character wildcard. The user can use the question mark to find values matching a pattern in which the number of characters is fixed, as in the following examples:

- Enter `Eriks?n` to find names like "`Erikson`" and "`Eriksen`."
- Enter `New??n` to find names like "`Newton`," "`Newman`," and "`Newson`," but not "`Newilsson`."

| The symbol **|** between values **a** and **b** represents the logical OR. The following entry specifies any of three numbers:

- *102 | 105 | 118*

[ ] When used in conjunction with the * and ? wildcard characters, the brackets enclose a list of characters, including ranges, to be matched.

^ A caret (^) as the first character within the brackets matches any character not listed. For example, the search value [^AB]* specifies all strings beginning with characters other than A or B.

- A hyphen (–) between characters within brackets specifies a range. The search value [^d-f*] specifies all strings beginning with characters other than lowercase d, e, or f. If you omit the * or ? wildcard, 4GL treats the brackets as literal characters, not as operators.

Searching for All Rows

If *none* of the fields contain search values when the user completes entry for the CONSTRUCT statement, 4GL uses " 1=1" as the Boolean expression. Notice that this string begins with a blank character. In a WHERE clause, this search criterion causes 4GL to select *all* the rows of the specified table(s).

Place a conditional statement after the CONSTRUCT statement to check for this expression, or to examine the field buffers in the AFTER CONSTRUCT control block, if you want to prevent users from selecting all rows. The next fragment, for example, tests for the " 1=1" expression. If this is found, the LET statement limits the resulting query list by creating a Boolean expression that searches only for customers with numbers less than or equal to 110:

```
CONSTRUCT BY NAME query_1 ON customer.*
IF query_1 = " 1=1" THEN
    LET query_1 = " customer_num <= 110"
    MESSAGE "You entered nothing. Here are customers ",
            "with codes less than 111."
    SLEEP 3
END IF
```

The FIELD_TOUCHED( ) operator (page 4-65) describes an equivalent test.

Positioning the Screen Cursor

When the user presses RETURN or TAB, the screen cursor moves from one field to the next in the order specified in the FROM clause, or implied by the column list in the BY NAME clause. The user can also press the following *arrow keys* to alter this behavior and to position the cursor explicitly:

Key	Effect on Cursor
[↓]	By default, it moves to the *next* field. If you specify the FIELD ORDER UNCONSTRAINED option of the OPTIONS statement, this key moves the cursor to the field *below* the current field.
[↑]	By default, it moves to the *previous* field. If you specify the FIELD ORDER UNCONSTRAINED option of the OPTIONS statement, this key moves the cursor to the field *above* the current field.
[←]	It moves one space to the *left* within a field. If this is the beginning of the field, the cursor moves to the beginning of the *previous* field.
[→]	It moves one space to the *right* within a field. If this is the end of the field, **4GL** creates a workspace at the bottom of the screen, and places the cursor there, so the user can continue entering values.

These arrow keys all operate non-destructively. That is, they move the screen cursor without erasing any underlying character.

When the cursor moves to a new field, the CONSTRUCT statement clears the Comment line and the Error line. The Comment line displays the text defined with the COMMENTS attribute in the form specification file. The Error line displays system error messages, output from the built-in ERR_PRINT( ) and ERR_QUIT( ) functions, and ERROR statement messages.

If the user enters search criteria that exceed the length of the screen field, 4GL automatically moves the cursor down to the Comment line and allows the user to continue entry. When the user presses RETURN or TAB, 4GL clears the Comment line. The field buffer contains all the criteria that the user entered, even though only a portion is visible in the screen display.

In a multiple-segment field (that is, one with the WORDWRAP attribute), 4GL ignores any values that the user enters in any segments beyond the *first* segment of the field. Similarly, in a screen array, the user can enter criteria only in the first screen record of the array during a CONSTRUCT statement.

Editing During a CONSTRUCT Statement

The user can employ these keys for editing during a CONSTRUCT statement:

Key	Effect
CONTROL-A	Toggles between insert and typeover mode.
CONTROL-D	Deletes characters from the cursor position to the end of the field.
CONTROL-H	Moves the cursor non-destructively one space to the left within a field. This is equivalent to pressing the [←] key.
CONTROL-L	Moves the cursor non-destructively one space to the right within a field. This is equivalent to pressing the [→] key.
CONTROL-R	Redisplays the screen.
CONTROL-X	Deletes the character beneath the cursor.

Completing a Query

The following actions terminate the CONSTRUCT statement:

- The user presses one of the following keys:
  - The Accept key.
  - The RETURN or TAB key from the last field (when INPUT WRAP is not set in the OPTIONS statement).
  - The Interrupt or Quit key.
- 4GL executes the EXIT CONSTRUCT statement.

The user must choose Accept to complete the query under these conditions:

- INPUT WRAP is specified in the OPTIONS statement.
- An AFTER FIELD block for the last field includes a NEXT FIELD clause.

By default, the Accept, Cancel, Interrupt, or Quit terminates both the query and the CONSTRUCT statement. (But pressing the Interrupt or Quit key can also immediately terminate the program, unless the program has also executed the DEFER INTERRUPT and DEFER QUIT statements.)

If 4GL previously executed a DEFER INTERRUPT statement in the program, an Interrupt causes 4GL to do the following:

- Set the global variable **int_flag** to a non-zero value.
- Terminate the CONSTRUCT statement, but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, the Quit key causes 4GL to do the following:

- Set the global variable **quit_flag** to a non-zero value.
- Terminate the CONSTRUCT statement, but not the 4GL program.

When the user terminates a CONSTRUCT statement, 4GL executes the statements in the AFTER CONSTRUCT clause. (If the CONSTRUCT statement is terminated by an EXIT CONSTRUCT statement, however, 4GL does not execute the statements in the AFTER CONSTRUCT clause.) In these cases, the statements in the AFTER FIELD clause of the current field are not executed. When a NEXT FIELD clause appears in either of these clauses, 4GL ignores the Accept keystroke, and focus moves to the specified field.

The following program segment uses a simple CONSTRUCT statement to specify the search condition of a WHERE clause. The variable **query_1** is declared as CHAR(250), and the **cursor_1** cursor executes the query.

```
CONSTRUCT BY NAME query_1
    ON order_num, customer_num, order_date, ship_date
    ATTRIBUTE(BOLD)

LET s1 = "SELECT * FROM orders
    WHERE ", query_1 CLIPPED,
    " ORDER BY order_date, order_num"
PREPARE s_1 FROM s1
DECLARE cursor_1 CURSOR FOR s_1
FOREACH cursor_1 INTO order_rec.*
    ...
END FOREACH
```

The following program fragment demonstrates several CONSTRUCT form management blocks:

```
CONSTRUCT BY NAME query_1 ON customer.*

    BEFORE CONSTRUCT
        MESSAGE "Enter search criteria; ",
            "press ESC to begin search."
        DISPLAY "Press F1 or CTRL-W for field help." AT 2,1

    ON KEY (F1, CONTROL-W)
        CALL customer_help()  -- display field level help

    BEFORE FIELD state
        MESSAGE "Press F2 or CTRL-B ",
            "to display a list of states."

    ON KEY (F2, CONTROL-B)
        IF INFIELD(state) THEN
            CALL statehelp()   -- display list of states
        END IF

    AFTER FIELD state
        MESSAGE "Enter search criteria; ",
            "press ESC to begin search."

    AFTER CONSTRUCT     -- check for blank search criteria
        IF NOT FIELD_TOUCHED(customer.*) THEN
            PROMPT "Do you really want to see ",
                "all customer rows? (y/n) "
                FOR CHAR answer
            IF answer MATCHES "[Nn]" THEN
                MESSAGE "Enter search criteria; ",
                    "press ESC to begin search."
                CONTINUE CONSTRUCT -- reenter query by example
            END IF
        END IF
END CONSTRUCT

LET s1 = "SELECT * FROM customer WHERE ", query_1 CLIPPED
PREPARE s_1 FROM s1
DECLARE q_curs CURSOR FOR s_1
DISPLAY "" AT 2,1                -- clear line 2 of text
LET exist = 0
```

References

DECLARE, DEFER, DISPLAY FORM, EXECUTE, LET, OPEN FORM, OPEN WINDOW, OPTIONS, SELECT, PREPARE

CONTINUE

The CONTINUE keyword transfers control of execution from a statement block to another location in the currently executing compound statement.

```
CONTINUE keyword ─────────────────────────────────────┤
```

keyword is the name of the current 4GL statement. You can choose from the following keywords: CONSTRUCT, FOR, FOREACH, INPUT, MENU, or WHILE.

Usage

You can use CONTINUE within a statement block of the currently executing compound statement that *keyword* specifies. This is a run-time instruction to transfer control within the current statement. (Use the EXIT keyword, rather than CONTINUE, to terminate the compound statement unconditionally.)

Page 3-285 describes the effect of CONTINUE in the WHENEVER statement.

CONTINUE in CONSTRUCT, INPUT, and INPUT ARRAY Control Blocks

The CONTINUE CONSTRUCT (page 3-47) and CONTINUE INPUT statements (page 3-169) cause 4GL to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form, giving the user another chance to enter data in that field.

CONTINUE in FOR, FOREACH, and WHILE Loops

The CONTINUE FOR (page 3-103), CONTINUE FOREACH (page 3-108), or CONTINUE WHILE (page 3-288) cause the current FOR, FOREACH, or WHILE loop (respectively) to begin a new cycle immediately. If conditions do not permit a new cycle, however, the looping statement terminates.

CONTINUE in MENU Control Blocks

The CONTINUE MENU statement causes 4GL to ignore the remaining statements in the current MENU control block, and redisplay the menu. The user can then choose another menu option. (See “MENU” on page 3-193.)

References

CONSTRUCT, FOR, FOREACH, GOTO, INPUT, INPUT ARRAY, MENU, WHILE, WHENEVER

CURRENT WINDOW

The CURRENT WINDOW statement makes a specified 4GL window the current window (that is, the topmost 4GL window in the window stack).

```
CURRENT WINDOW IS ──┬── window ──┬──┤
                    └── SCREEN ──┘
```

window is the name of the open 4GL window to be made current.

Usage

4GL maintains a list or "stack" of all open 4GL windows in the 4GL screen. The OPEN WINDOW statement creates a new 4GL window that is added to the top of this window stack, becoming the *current window.* When you close a 4GL window, that 4GL window is removed from the stack. The topmost 4GL window among those that remain open becomes the new current window. Its values take effect for the positions of reserved lines like Prompt, Message, Form, and Comment lines.

The current 4GL window is always completely visible, and can obscure all or part of any inactive 4GL windows. When you specify a new current window, 4GL adjusts the window stack by moving the new current 4GL window to the top, and closing the gap in the stack left by this 4GL window.

Programs with multiple 4GL windows may need to switch to a different current window so that input and output occur in the appropriate 4GL window. To make a 4GL window the current window, specify its name in the CURRENT WINDOW statement. For example, the following statement makes **win1** the current 4GL window:

```
CURRENT WINDOW IS win1
```

When a program starts, the 4GL screen is the current 4GL window. Its name is SCREEN. To make this the current 4GL window, specify the keyword SCREEN instead of a *window* identifier:

```
CURRENT WINDOW IS SCREEN
```

If a 4GL window contains a form, that form becomes the current form when a CURRENT WINDOW statement specifies the name of that 4GL window.

The CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, and MENU statements use only the current 4GL window for input and output. If the user displays another form (for example, through an ON KEY clause) in one of these statements, the 4GL window containing the new form

becomes the current window. When the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or MENU statement resumes, its original 4GL window becomes the current window.

The next program fragment opens multiple 4GL windows, including one called **w2**. Interactive statements that use 4GL window **w2** can follow the CURRENT WINDOW statement within the **do2( )** function. If function **do2( )** terminates after assigning modular variable **next_win** any value but 2, the CALL statement in the WHILE loop invokes a different function. The **w2** 4GL window remains current until another CURRENT WINDOW statement specifies some other 4GL window, or until CLOSE WINDOW **w2** is executed.

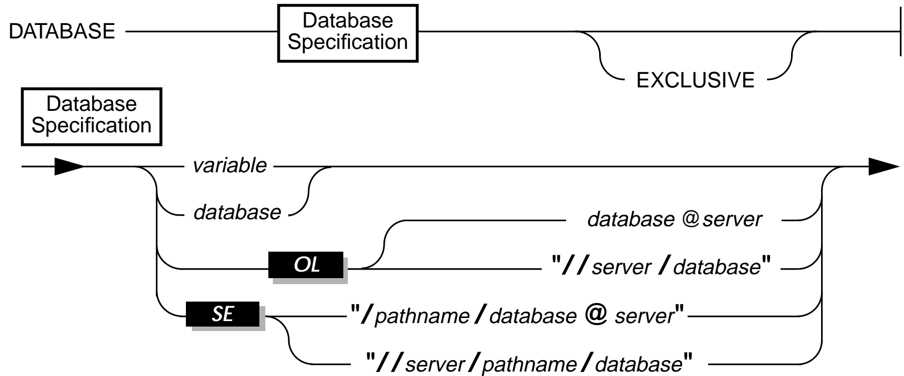
```
DEFINE next_win INTEGER
MAIN
OPEN WINDOW w1 AT 3,3 WITH FORM "cust1"
OPEN WINDOW w2 AT 9,15 WITH FORM "cust2"
OPEN WINDOW w3 AT 15,27 WITH FORM "cust3"
    . . .
    LET next_win = 1
    WHILE next_win IS NOT NULL
        CASE (next_win)
        WHEN 1
            CALL do1()
        WHEN 2
            CALL do2()
        WHEN 3
            CALL do3()
    . . .
        END CASE
    END WHILE
CLOSE WINDOW w1
CLOSE WINDOW w2
CLOSE WINDOW w3
    . . .
END MAIN
FUNCTION do2()
    LET next_win = NULL
    CURRENT WINDOW IS w2
     . . .
END FUNCTION
```

References

CLEAR, CLOSE WINDOW, DISPLAY ARRAY, INPUT, INPUT ARRAY, MENU, OPEN WINDOW, OPTIONS

DATABASE

The DATABASE statement opens a default database at compile time, or a current database at run time. (See also the description of the DATABASE statement in the *Informix Guide to SQL: Syntax*, Version 6.0.)



database is the name of a database. Blank spaces are not valid between quotes nor after the @ symbol.

pathname is the directory path to the parent directory of the **.dbs** directory.

server is the name of the host system where *database* resides.

variable is a variable that contains the database specification (page 3-59). (You can specify a *variable* only in a MAIN or FUNCTION block.)

Usage

This statement is not required if your **4GL** program does not reference entities in a database. You can use the DATABASE statement in two distinct ways, depending on the context of the statement within its source code module:

- To specify the *default* database for the compiler to use in declaring data types indirectly in DEFINE statements (page 3-59), or for INITIALIZE (page 3-125) or VALIDATE (page 3-278) to access **syscolatt** or **syscolval**. Specifying a default database also results in that database being opened automatically at run time.
- To specify the *current* database at run time, so that SQL statements can access data and other entities in that database (page 3-60).

The Database Specification

The DATABASE statement can specify any accessible database on the current **INFORMIX-OnLine** database server (or on another **OnLine** server, if you also specify its name). This becomes the *default database* at compile time, and the *current database* at run time. To reference entities in any other database, you must use the CLOSE DATABASE statement and then another DATABASE statement or table qualifiers (page 3-361).

The DATABASE statement closes any other open database on the same database server. If a database is open on another server, you must first use CLOSE DATABASE explicitly to close that current database, or an error occurs. An error results if you specify a database that **4GL** cannot locate or cannot open or for which the user of your program does not have access privileges.

SE

Only the databases stored in your current directory, or in a directory specified in your DBPATH environment variable, are recognized.

To specify a database that does not reside in your current directory nor in a DBPATH directory, you must follow the DATABASE keyword with a complete pathname, or with a program variable that evaluates to the full pathname of the database (excluding the **.dbs** extension).

NLS

When the database connection is established between a user session and the database server by way of the DATABASE statement with NLS features active, the information in environment variables LC_CTYPE and LC_COLLATE is transmitted with the request for database service. The database server uses the information to check that the user locale and existing database locale match. If they do not match, the request for database service is rejected. This process is referred to as *locale consistency checking.*

LC_CTYPE and LC_COLLATE values active at the time of database creation are stored with the database in a system table. These values are kept unchanged throughout the life of the database to ensure the consistent use of collating sequences, codesets, and formatting rules. The character set and collation settings for a database cannot be changed; the data must be unloaded and reloaded into a different database to change locales.
For complete information on using NLS, see Appendix E.

The Default Database at Compile Time

The DEFINE statement (page 3-69) can specify that a record is LIKE a table, or that a variable is LIKE a column in a database table. Even if you qualify the name of the table with a database name, this requires a DATABASE statement

to specify a default database at compile time. The compiler looks in this default database for the schema of tables whose columns are to be used as templates for declaring variables indirectly through the LIKE keyword.

To declare variables indirectly, the DATABASE statement must precede any program block in each module that includes a DEFINE . . . LIKE declaration, and must precede any GLOBALS . . . END GLOBALS statement (page 3-117). It must also precede any DEFINE . . . LIKE declaration of module variables. Here the *database* name must be expressed explicitly, and not as a variable. The EXCLUSIVE keyword is not valid in this context. (The INITIALIZE . . . LIKE and VALIDATE . . . LIKE statements likewise require that DATABASE specify a default database before the first program block in the same module.)

If you want different program blocks to use the same database, you can repeat the same DATABASE statement in each program block in which entities in the database are referenced or created. Alternatively, you can create a file that includes only the DATABASE and the GLOBALS . . . END GLOBALS (page 3-117) statements, and then include GLOBALS "*filename*" statements at the beginning of each module that requires the DATABASE statement.

The next example shows the contents of a file in which no global variables are declared, but the **zeitung** database can be accessed by statements in any other source modules that include the GLOBALS "*filename*" statement:

```
DATABASE zeitung
GLOBALS
END GLOBALS
```

Here GLOBALS . . . END GLOBALS can also include DEFINE statements.

The Current Database at Run Time

If your **4GL** program is designed to interact with a database at run-time, the DATABASE must specify a current database that subsequent SQL statements can reference, until it is closed (by CLOSE DATABASE or by another DATABASE statement, for example), or until the program terminates.

In this case, the DATABASE statement must occur in a FUNCTION or the MAIN program block, and must follow any DEFINE statements in that block, or else it must precede the MAIN program block. When DATABASE specifies the current database, the database specification can be in a **4GL** variable, and you can include the EXCLUSIVE keyword (page 3-61).

If a DATABASE statement (or a GLOBALS "*filename*" statement, where *filename* includes the DATABASE statement) precedes the MAIN statement, then the **4GL** compiler (in effect) inserts the same DATABASE statement into the

beginning of the MAIN program block, before the first executable statement, if no other DATABASE statement precedes MAIN. In this special case, the same DATABASE statement produces both compile-time and run-time effects.

You cannot include the DATABASE statement within a REPORT program block. If a **4GL** report definition requires a two-pass report (page 6-22), an error occurs if no database is open when the report is run, or if the report driver issues a DATABASE statement while the report is running (page 3-101).

You cannot include the DATABASE statement in a multiple-statement PREPARE operation. (See also the descriptions of the PREPARE statement and of the CLOSE DATABASE statement in the *Informix Guide to SQL: Reference*.)

The EXCLUSIVE Keyword

The DATABASE statement with the EXCLUSIVE keyword opens the database in exclusive mode but prevents access by anyone but the current user. It is valid only in a FUNCTION or MAIN program block. To allow others to access a database that was opened in EXCLUSIVE mode, you must execute the CLOSE DATABASE statement. Then use DATABASE without the EXCLUSIVE keyword to reopen the database, if appropriate.

The following statement opens the **stores2** database on the **mercado** server in exclusive mode:

```
DATABASE stores2@mercado EXCLUSIVE
```

If another user already has the specified database open, exclusive access is denied, an error is returned, and no database is opened.

Testing SQLCA.SQLAWARN

You can determine the type of database that the DATABASE statement opens by examining the built-in SQLCA.SQLAWARN variable (page 2-23) after the DATABASE statement has executed successfully:

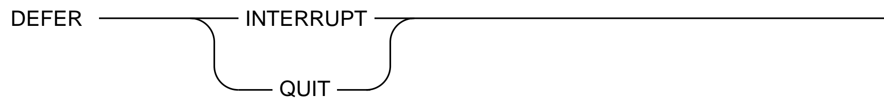
- If the specified database uses transactions, then SQLCA.SQLAWARN[2], the second element of the SQLCA.SQLAWARN global record, contains a `W`.
- If the database is ANSI-compliant, then SQLCA.SQLAWARN[3], the third element of the SQLCA.SQLAWARN global record, contains a `W`.
- If **INFORMIX-OnLine** is the database engine, then SQLCA.SQLAWARN[4], the fourth element of the SQLCA.SQLAWARN global record, contains a `W`.

References

DEFINE, FUNCTION, GLOBALS, INITIALIZE, MAIN, REPORT, VALIDATE

DEFER

The DEFER statement prevents 4GL from terminating program execution when the user presses the Interrupt key or the Quit key.



Usage

DEFER is a method of intercepting asynchronous signals from outside the program.

Unless it includes the DEFER statement, the 4GL application terminates whenever the user presses the Interrupt or Quit key. The Interrupt key is CONTROL-C, and the Quit key is CONTROL-\.

The DEFER statement tells 4GL to set a built-in global variable to a non-zero value, rather than terminate, when the user presses one of these keys:

- If the user presses the Interrupt key when DEFER INTERRUPT has been specified, 4GL sets the built-in global variable **int_flag** to TRUE.
- If the user presses the Quit key when DEFER QUIT has been specified, 4GL sets the built-in global variable **quit_flag** to TRUE.

The DEFER INTERRUPT and DEFER QUIT statements can appear only in the MAIN program block, and only once there in any program. Once executed, the DEFER statement remains in effect for the duration of the program; you cannot restore the original function of the Interrupt key or the Quit key.

4GL programs can include code to check whether **int_flag** or **quit_flag** is TRUE, and if so, to take appropriate action. Be sure also to reset **int_flag** or **quit_flag** to FALSE (that is, to zero) so that subsequent tests are valid.

Interrupting Screen Interaction Statements

If DEFER INTERRUPT has executed, you can specify INTERRUPT to make the Interrupt key the activation key in an ON KEY clause of CONSTRUCT, INPUT ARRAY, or INPUT statements. If the user presses the Interrupt key, control returns to the same field, unless the statement block includes the EXIT or NEXT FIELD keywords. Without the ON KEY (INTERRUPT) specification, an Interrupt signal transfers control to the AFTER INPUT or AFTER CONSTRUCT control block, if these are present, or else to END INPUT or END CONSTRUCT. Any AFTER FIELD clause for the current field is ignored, and the **int_flag** is

reset to TRUE. (After DEFER QUIT, pressing the Quit key resets the **quit_flag** to TRUE, but the Quit key has no effect on CONSTRUCT, INPUT ARRAY, or INPUT statements.)

To make sure that **int_flag** or **quit_flag** is reset, you can use the LET statement to set both variables to FALSE immediately before the CONSTRUCT, DISPLAY ARRAY, INPUT, MENU, or PROMPT statements. After DEFER INTERRUPT has executed, if the user presses the Interrupt key during any DISPLAY ARRAY or PROMPT statement, program control leaves the current statement, and 4GL sets the **int_flag** to a non-zero value. (When a MENU statement is executing, however, program control remains in the MENU statement.)

To have the user terminate a statement with a key other than the Interrupt key, use the ON KEY clause to define the action of the desired key sequence.

The next program fragment executes the DEFER INTERRUPT statement in the MAIN program block, and then calls a function that prompts the user to enter criteria for retrieving data from the **stock** table.

```
MAIN
    . . .
    DEFER INTERRUPT
    . . .
    CALL find_stock()
    . . .
END MAIN

FUNCTION find_stock()
    DEFINE
        where_clause CHAR(200)
    . . .
    DISPLAY "Enter selection criteria for ",
            "the stock items you want." AT 10,1
    LET int_flag = FALSE
    CONSTRUCT BY NAME where_clause
            ON stock.* FROM s_stock.*
    IF int_flag THEN
        ERROR "Query cancelled."
        RETURN
    END IF
    . . .
END FUNCTION
```

If the user decides not to enter any selection criteria, pressing the Interrupt key terminates the CONSTRUCT statement without executing the query.

If **int_flag** flag is set to a non-zero value (TRUE), the program terminates the function by executing a RETURN statement. Notice that the function resets the value of **int_flag** to FALSE (zero) before beginning the CONSTRUCT.

Here if **int_flag** is set to a non-zero value (evaluates to TRUE), a RETURN statement terminates the function. Notice that in this example, the **find_stock( )** function explicitly resets the value of **int_flag** to FALSE (zero) before beginning the CONSTRUCT statement.

Interrupting SQL Statements

To enable the Interrupt key to interrupt SQL statements, your program must contain:

- The DEFER INTERRUPT statement.
- The OPTIONS statement with the SQL INTERRUPT ON option.

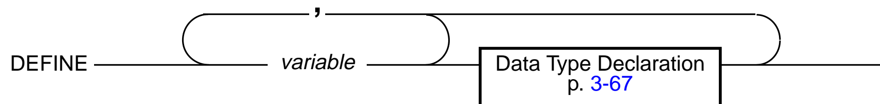
The keywords SQL INTERRUPT OFF restore the default of uninterruptable SQL statements. The section "Interrupting SQL Statements" on page 3-235 describes this feature in detail and its effect on the current database transaction.

References

CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, MAIN, MENU, OPTIONS, PROMPT, WHENEVER

DEFINE

The DEFINE statement declares the names and data types of 4GL variables.



variable is a name that you declare here as the identifier of a variable.

Usage

A variable is a named location in memory that can store a single value, or an ordered set of values. Except for predefined global variables like **status**, **int_flag**, **quit_flag**, or the **SQLCA** record, you cannot reference a variable before it has been declared by the DEFINE statement.

Sections that follow describe the following topics:

Topic	Page
The Context of DEFINE Declarations	3-65
Global Variables in Multiple-Module Programs	3-66
Declaring the Names and Data Types of Variables	3-67
Variables of Simple Data Types	3-68
Indirect Typing	3-69
Variables of Large Data Types	3-70
Variables of Structured Data Types	3-70

The Context of DEFINE Declarations

The DEFINE statement declares the identifier of at least one variable. There are two important things to know about these identifiers:

- Where in the program can they be used? The answer defines the *scope of reference* of the variable. A point in the program where an identifier can be used is said to be *in* the scope of the identifier. A point where the identifier is not known is *outside* the scope of the identifier.
- When is *storage* for the variable allocated? Storage can be allocated either *statically* when the program is loaded to run, (at load time), or *dynamically* while the program is executing (at run time).

The context of its declaration in the source module determines where a variable can be referenced by other **4GL** statements, and when storage is allocated for the variable in memory.

The DEFINE statement can appear in only two contexts:

1. Within a FUNCTION, MAIN, or REPORT program block, DEFINE declares *local* variables, and causes memory to be allocated for them. These DEFINE declarations of local variables must precede any executable statements within the same FUNCTION, REPORT, or MAIN program block.
   - The scope of reference of a local variable is restricted to the same program block. Elsewhere, the variable is not visible.
   - Storage for local variables is allocated when its FUNCTION, REPORT, or MAIN block is entered during execution. Functions can be called recursively, and each recursive entry creates its own set of local variables. The variable is unique to that invocation of its program block. Each time the block is entered, a new copy of the variable is created.
2. Outside any FUNCTION, REPORT, or MAIN program block, the DEFINE statement declares names and data types of *module* variables, and causes storage to be allocated for them. These declarations must appear before, any program blocks.
   - Scope of reference is from the DEFINE statement to the end of the same source code module (but the variable is not visible within this scope in program blocks where a local variable has the same identifier).
   - Memory storage for variables of *module* scope is allocated statically, in the executable image of the program.

Global Variables in Multiple-Module Programs

The GLOBALS statement can extend the visibility of a module-scope variable to additional modules. GLOBALS (page 3-117) can be used in two ways:

- GLOBALS DEFINE . . . END GLOBALS declares global variables directly.
- GLOBALS "*filename*" indirectly extends to the current source module the scope of the variables that were declared as global in *filename*.

The following example declares global variables directly in a source file:

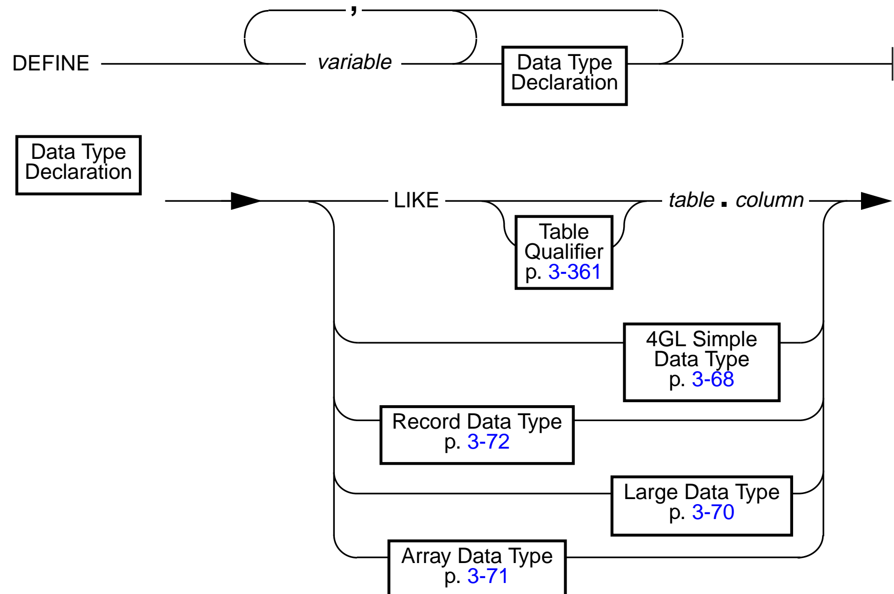
```
GLOBALS
    DEFINE a,b,c INT,
           x,y,z CHAR(10)
END GLOBALS
```

If this GLOBALS statement were in a source module called **gloFile.4gl** that included no other statement, then you could extend the scope of these six variables to any other module that contained the declaration:

```
    GLOBALS "gloFile.4gl"
```

Declaring the Names and Data Types of Variables

The DEFINE statement must declare the *name* and the *data type* of each new variable, either explicitly or else implicitly (by using the LIKE keyword):



column is the name of a database *column*, qualified by the name or by the synonym of a table or view.

table is the name or synonym of a database table or view.

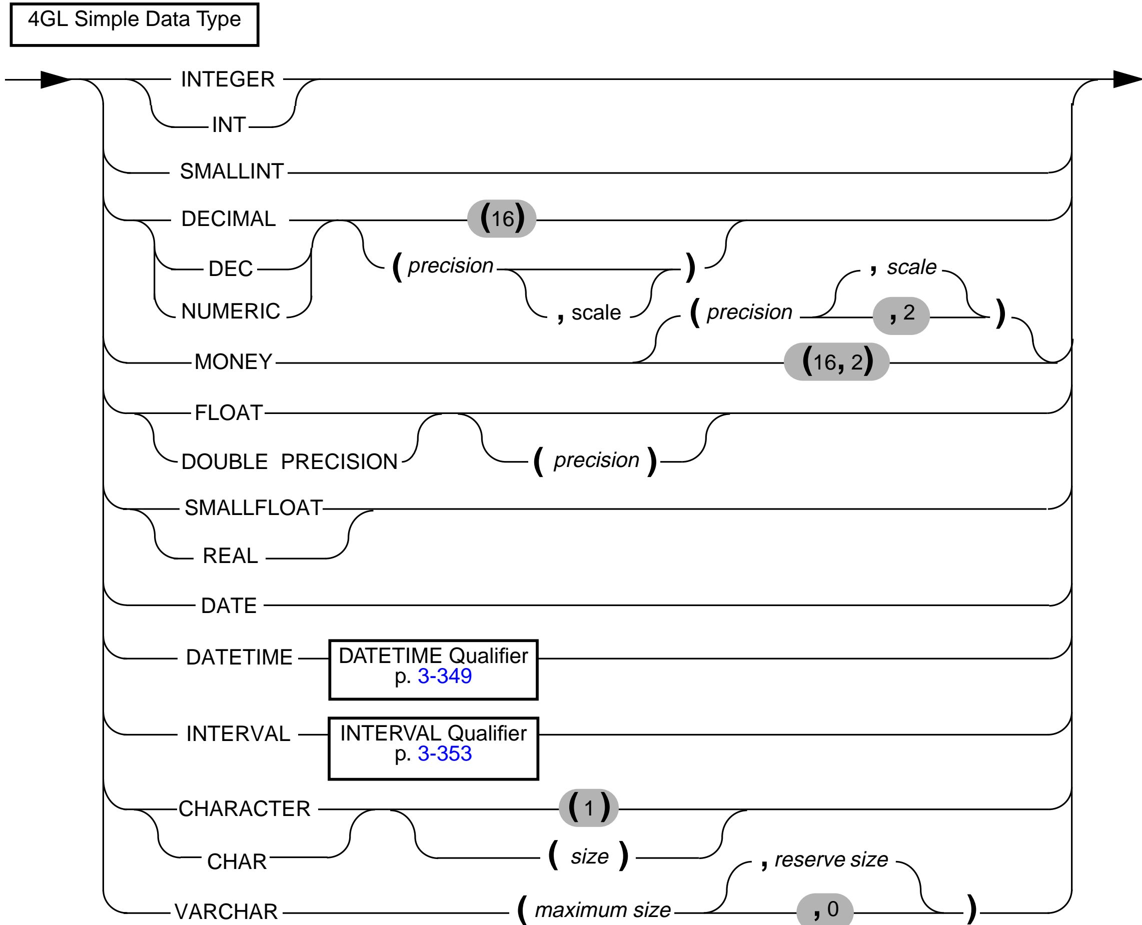
variable is the name of the variable. This must be unique among variables within the same scope of reference (page 3-65).

See the section "Data Types of 4GL" on page 3-293 for details of the various data types that you can specify when you declare **4GL** variables.

See the section "Indirect Typing" on page 3-69 for details of using the LIKE keyword (and the DATABASE statement, page 3-58) to declare **4GL** variables.

In the **Rapid Development System**, the compiler supports no more than 32,767 variables in a single program, and no more than 64,535 characters in all the names of **4GL** variables, including record members. The **C Compiler Version** has no such limits, apart from the memory capacity of your system.

Variables of Simple Data Types

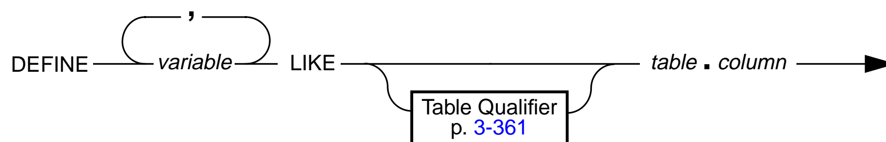


maximum size	is a whole number from 1 to 255, specifying the largest number of characters that the VARCHAR data type can store.
precision	specifies the total number of decimal digits, from 1 to 32.
reserve size	is a whole number, from 0 to *maximum size*. The default is 0. (Like *precision* for FLOAT or DOUBLE PRECISION, **4GL** accepts this value for compatibility with SQL, but does not use this value.)
scale	is a whole number, from 1 to *precision*, specifying the number of digits in the fractional part of a fixed-point number.
size	is a whole number from 1 to 32,767, specifying how many characters a CHAR data type can store. The default is 1.

All these declaration parameters are literal integers (page 3-340).

Indirect Typing

You can use the LIKE keyword to declare a variable that has the same simple, BYTE, or TEXT data type as a specified column in a database table.



column is the identifier of some column in *table*, as it appears in the **syscolumns** table of the system catalog.

table is the identifier or synonym of a table or view in the default database that was specified in the DATABASE statement.

variable is the **4GL** identifier of a variable that you declare here.

If *table* is a view, then *column* cannot be based on an aggregate. If LIKE references a SERIAL column, the new variable is of the INTEGER data type.

The DATABASE statement must specify a default database (page 3-59) before the first program block (or before the first DEFINE statement that uses LIKE to define module-scope or global variables) in the current module. At compile time, **4GL** substitutes a data type for the LIKE declaration, based on the schema of *table*. (If that schema is subsequently modified, recompile the module to restore the correspondence between variables and columns.)

Any *column* in the LIKE declaration has either a simple (page 3-68) or a large (page 3-70) data type. (An OnLine database cannot include ARRAY nor RECORD columns. An SE database cannot include ARRAY, BYTE, RECORD, TEXT, nor VARCHAR columns.)

The table qualifier must specify *owner* if *table.column* is not a unique column identifier within its database, or if the database is ANSI-compliant and any user of your **4GL** application is not the owner of *table*.

See also "RECORD Variables" on page 3-72, which shows how the LIKE keyword in RECORD declarations can declare *implicit names* (the same as *column* names) for member variables of a record, and can indirectly assign to these member variables the data types of the corresponding database columns.

In the demonstration database, the **manufact** table has three columns:

- **manu_code** of data type CHAR(3)
- **manu_name** of data type CHAR(15)

- **lead_time** of data type INTERVAL DAY(3) TO DAY

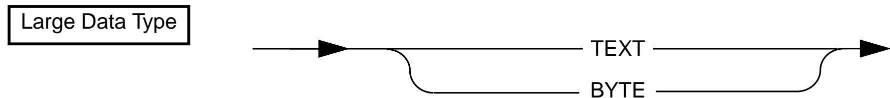
The following declarations of variables are based on the **manufact** table:

```
DATABASE demo5
DEFINE codename RECORD LIKE manufact.*
    -- equivalent to "manu_code char(3), manu_name char(15),
    --               lead_time interval day(3) to day"
    DEFINE hidden LIKE manufact.manu_code
    -- equivalent to "hidden char(3)"
    DEFINE leaden LIKE manufact.lead_time
    -- equivalent to "lead_time interval day(3) to day"
```

Variables of Large Data Types

These store pointers to *binary large object* (blob) values, up to 2^{31} bytes in size (or up to a limit imposed by your implementation of **INFORMIX-OnLine**):

- TEXT Character strings.
- BYTE Any data, including binary, that can be stored on your system.



Unlike BYTE and TEXT declarations in SQL, there is no IN clause in DEFINE statements; in 4GL the LOCATE statement (page 3-186) supports the functionality of the IN clause.

The CALL and RUN statements cannot include the BYTE nor TEXT keywords in their RETURNING clauses. See "Data Types of 4GL" for more information about the BYTE (page 3-298) and TEXT (page 3-317) data types.

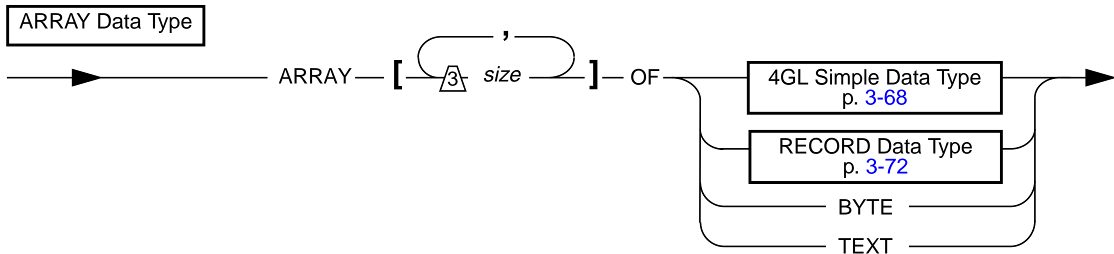
Variables of Structured Data Types

INFORMIX-4GL supports two *structured* data types for storing sets of values:

- ARRAY Arrays of values of any data type except ARRAY.
- RECORD Sets of values of any data type, or any combination of types.

ARRAY Variables

The ARRAY keyword declares a structured variable that can store a 1-, 2-, or 3-dimensional array of values, all of the same data type:



size is the number (up to 32,767) of elements in a dimension. Dimensions can be different sizes, up to the limit of your C compiler.

The elements of an ARRAY variable can be of any data type except ARRAY. The limit on the total number of elements in an array is compiler-dependent.

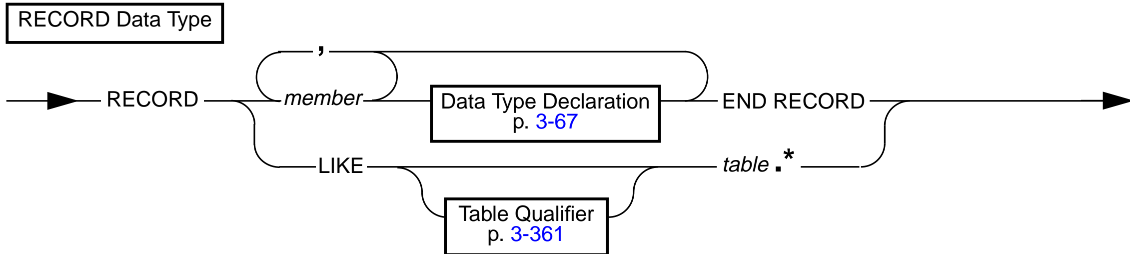
Although the **Rapid Development System** compiler supports no more than 32,767 variables in one program, an array counts as only a single variable in respect to this limit, regardless of the number of array elements. Array elements can be of any **4GL** data type except ARRAY, but an element can be a record that contains an array member.

You cannot specify an ARRAY data type as an argument nor as a returned value of a **4GL** function. The CALL and RUN statements cannot include the ARRAY keyword in their RETURNING clauses. In the DEFINE section of a REPORT statement, formal arguments (page 6-8) cannot be declared as ARRAY data types, nor as RECORD variables that contain ARRAY members. (Data types of local variables that are not formal arguments are unrestricted.)

A database table cannot include a column of the ARRAY nor RECORD data types, because the SQL language does not support structured data types.

RECORD Variables

A **4GL** *program record* is a collection of members, each of which is a variable. The member variables of a record can be of any **4GL** data type, including the simple data types (page 3-68), the structured (ARRAY, RECORD) data types, and the large (BYTE, and TEXT) data types.



member is a name that you declare for a member variable of the record; this identifier must be unique within the record.

table is the identifier or synonym of a table or view in the default database that was specified in the DATABASE statement.

The DATABASE statement must specify a default database (page 3-59) before the first program block (or before the first DEFINE statement that uses LIKE to define module-scope or global variables) in the current module.

Specify LIKE *table*.* to declare the record members implicitly, with identifiers and data types that correspond to all the non-SERIAL columns of *table*. You do not need the END RECORD keywords to declare a single record whose members correspond to all the non-SERIAL columns of *table*:

```
recordname RECORD LIKE table.*
```

In this context, *table*`.*` cannot be a view containing an aggregate column.

You can use multiple LIKE clauses in the same RECORD declaration, provided that the LIKE keyword does not immediately follow the keyword RECORD:

```
DEFINE cust_ord_item
    RECORD
        cust_no LIKE customer.customer_num,
        ord RECORD LIKE orders.*,-- row from "orders" table
        it1 RECORD a1 LIKE items.item_num, -- subset of row
                   b1 LIKE items.order_num -- in "items"
            END RECORD
        item_quantity LIKE items.quantity, --an "items" column
        it2 RECORD a2 LIKE items.total_price -- columns from
                   b2 LIKE stock.unit,       -- various tables
                   c2 LIKE manufact.manu_name
            END RECORD
    END RECORD
```

A compilation error occurs, however, if a LIKE clause begins the declaration of a record that is terminated by the END RECORD keywords. To declare a record with members that mirror the data types of a database table, but that also contains other members, declare one or more of the other members first. Then you can mix LIKE clauses and explicit variable declarations to the end of the record, as in the previous example.

Join columns often have the same name, but you must avoid the repetition of column names when using two or more LIKE clauses in the same scope of reference, so that both variables do not have the same name. In the demonstration database, both the **orders** and **items** tables include a column **order_num** that can join them. In the previous example, the record members declared LIKE the columns of **items** have the same sequence as in the table, but the record member that is declared like the second **order_num** column is called **item_order_num**.

Note: *A scroll cursor cannot be used with a record that has a BYTE or TEXT column.*

The section "Data Types of 4GL" describes ARRAY (page 3-297) and RECORD (page 3-313) data types. See also the INPUT ARRAY statement (page 3-152) and DISPLAY ARRAY statement (page 3-85) for information on using program arrays of records in interactive statements.

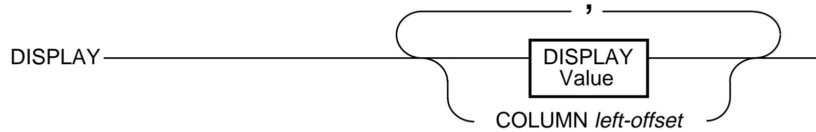
References

DATABASE, FUNCTION, GLOBALS, MAIN, REPORT

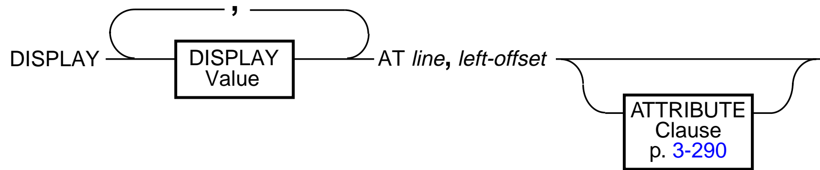
DISPLAY

The DISPLAY statement displays data values on the screen. (To produce output within a REPORT routine, you must use PRINT, rather than DISPLAY.) The syntax of DISPLAY determines whether output appears in a specified line of the current 4GL window or in a *form*.

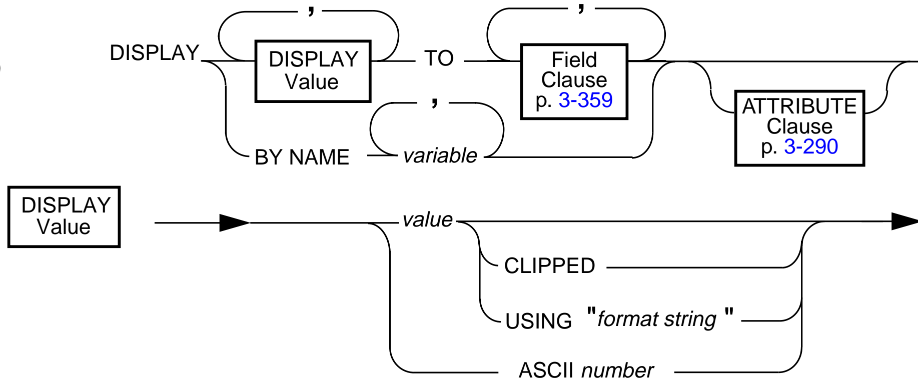
Case I:
(output in the
Line mode overlay)



Case II:
(in a specified line of
the current window)



Case III:
(in a screen form)



format string is a quoted string to specify a display format; see page 4-91.

left-offset is an integer variable or a literal integer, specifying the position of the first character of the next item of output within the current line of the screen or window; see page 4-40.

line is an integer variable or a literal integer, specifying the position of a *line* of the screen or of the current window.

number is a whole number, specifying an ASCII code to be displayed; see page 4-28 and Appendix G.

value is a quoted string, or the 4GL identifier of a named constant or variable, specifying a value to be displayed.

variable is the name of a variable that is also the name of a field.

Usage

The DISPLAY statement sends output directly to the screen, or to specified fields of a screen form (page 3-80). The DISPLAY statement *cannot* reference values of the ARRAY or BYTE data types. After DISPLAY is executed, changing the value of a displayed variable has no effect on the current display until you execute the DISPLAY statement again.

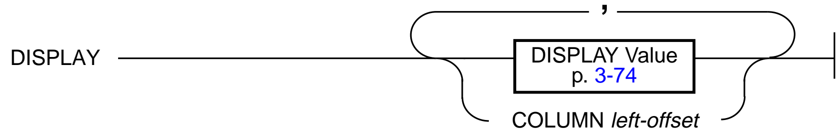
The following topics are described in this section:

Topic	Page
Sending Output to the Line Mode Overlay	3-76
Sending Output to the Current 4GL Window	3-77
Formatting Screen Output	3-77
The AT Clause	3-79
Sending Output to a Screen Form	3-80
The BY NAME Clause	3-81
The TO Clause	3-82
The ATTRIBUTE Clause	3-83
Displaying Numeric and Monetary Values	3-83

Sending Output to the Line Mode Overlay

The DISPLAY statement without qualifying clauses (or with the COLUMN operator) sends output to the Line mode overlay:

Case I:
(display output in
the Line mode overlay)



left-offset specifies the position of the first character of the next item of output within the Line mode overlay.

Interactive statements of 4GL produce screen output in either of two modes:

- *Formatted mode* statements: INPUT, INPUT ARRAY, CONSTRUCT, ERROR, MESSAGE, DISPLAY ARRAY, and DISPLAY (with any clause)
- *Line mode* statements: DISPLAY (without any clause)

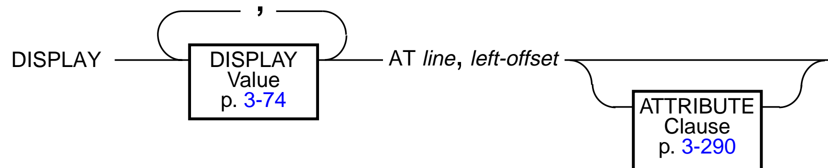
The PROMPT statement produces output in whichever of these two display modes is current. When it executes a DISPLAY statement that has no qualifying clause, **4GL** automatically opens a new 4GL window that covers the entire **4GL** screen until another interactive statement that produces Formatted mode output is encountered.

If the next interactive statement is neither a Line mode DISPLAY nor a PROMPT statement, the Line mode overlay disappears, revealing the **4GL** screen. Otherwise, any Line mode DISPLAY statement or PROMPT statement continues the display in the Line mode overlay.

Sending Output to the Current 4GL Window

If you include the AT keyword and specify coordinates, output is displayed, beginning in that location in the current **4GL** window:

Case II:
(to a specified line of current window)



left-offset is a literal integer (page 3-340) that specifies the position of the first character of the next item of output within the current line of the screen or window.

line is an integer expression (page 3-338) that returns a positive value, to specify a *line* of the current 4GL window (which can be the 4GL screen itself, if no other 4GL window is current).

Formatting Screen Output

The DISPLAY statement supports only a subset of the syntax of character expressions (page 3-343). You can use the **record.*** or the THROUGH or THRU notation (page 3-363) to reference the member variables of a record. You can refer to substrings of CHAR, VARCHAR, and TEXT variables by following the identifier with the starting and ending position of the substring, separated by a comma, and enclosed in brackets. For example, this statement displays characters 8 through 20 of the **full_name** variable:

```
DISPLAY "name", full_name[8,20],"added to database" AT 9, 2
```

You can use the following keywords to format the screen output:

- ASCII *number* (to display any ASCII character)
- CLIPPED (to truncate trailing blanks)
- COLUMN *number* (to begin the output at a specified character position)
- USING "*string*" (to format values of number or DATE data types)

Note: *You cannot use an* AT, ATTRIBUTE, BY NAME, *or* TO *clause with* COLUMN.

These operators are described in Chapter 4. No others are supported. If you want to display the current time, for example, you must assign the value of CURRENT to a program variable and then display that variable, rather than include the CURRENT operator among the list of DISPLAY values.

The following statement displays the values of two character variables in the format **lname, fname** on the next line, using the CLIPPED operator:

```
DISPLAY p_customer.lname CLIPPED, ", ", p_customer.fname
```

Unless you use the CLIPPED or USING operators, the DISPLAY statement formats character representations of the values of program variables and constants with display widths (including any sign) that depend on their declared data types, as the following chart indicates.

Data Type	Default Display Width (in characters)
CHAR	The length from the data-type declaration.
DATE	10.
DATETIME	From 2 to 25, as implied in the data-type declaration.
DECIMAL	(2 + *m*), for *m* the precision from the data-type declaration.
FLOAT	14.
INTEGER	11.
INTERVAL	From 3 to 25, as implied in the data-type declaration.
MONEY	(3 + *m*), for *m* the precision from the data-type declaration.
SMALLFLOAT	14.
SMALLINT	6.
VARCHAR	The maximum length from the data-type declaration.

When no *field* is referenced by the TO or BY NAME keywords, output begins on the screen where the AT *line*, *left-offset* coordinates position it, or (if the AT clause is omitted) it defaults to the line below the current cursor position.

Unless the COLUMN operator specifies a non-default character position, the output begins in the first character position, and successive output items within the same DISPLAY statement are *not* separated by blank spaces. For example, suppose the following program fragment runs on May 5, 1994:

```
DEFINE col INTEGER, cow DATE
LET col = 2
LET cow = CURRENT
DISPLAY COLUMN 3, "col", col, COLUMN 23, cow, cow
    SLEEP 5
```

This DISPLAY statement would produce one line of output on May 5, 1994:

left-offset = 3 left-offset = 23

```
  col          2        05/05/199405/05/1994
```

- Two blank spaces (the `COLUMN 3` specification)
- The string `col` (the "col" string specification)
- Ten blank spaces, followed by the character `2` (the `col` INTEGER variable)
- Seven blank spaces (the `COLUMN 23` specification)
- The string `05/05/199405/05/1994` (the `cow, cow` DATE variables)

Each DISPLAY statement begins its output on a new line. You can also use the AT clause to position output when no screen fields are specified by the TO or BY NAME clause. If no fields are specified, you cannot include an ATTRIBUTES clause in the DISPLAY statement, unless you also include the AT clause.

The AT Clause

You can use the AT clause to display text at a specified location in the current **4GL** window, which can be the **4GL** screen. The CLIPPED or USING operators can format the displayed values. You *cannot*, however, include the COLUMN operator in a DISPLAY statement that includes the AT clause.

The coordinates start with line 1 and character position 1 in the upper left corner of the **4GL** screen or the current 4GL window. The *line* values increase as you go down, and the character position values increase as you move from left to right. An error occurs if either coordinate value exceeds the dimensions of the **4GL** screen or the current 4GL window.

For example, the following DISPLAY statement displays the value of record member **total_price**, starting in line 22, at character position 5:

```
DISPLAY "TOTAL:  ", p_items.total_price AT 22, 5
```

Text that you display remains on the screen until you overwrite it. If you use the AT clause when the last element of variable list is a NULL CHAR value, 4GL clears to the end of the line. If you execute a Formatted-mode statement when Line mode output from a DISPLAY statement with no clause is visible, 4GL clears the screen or the current 4GL window before producing Formatted mode display. (Formatted mode statements include ERROR, MESSAGE, PROMPT, or DISPLAY with any AT, BY NAME, or TO clause.)

Do not use DISPLAY AT to display text where it could overwrite useful data. Because INPUT clears the Comment line and the Error line when the cursor moves between fields, it is often a good idea *not* to display text in the following positions of the current 4GL window or the 4GL screen:

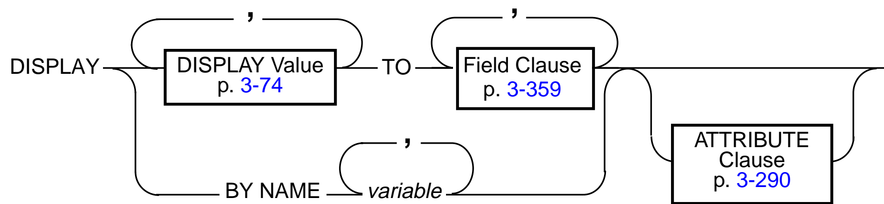
- The last line of the current 4GL window (the default Comment line).
- The last line of the 4GL screen (the default Error line).

If you want to use these lines for text display, make sure to reposition the Comment and Error lines. The OPEN WINDOW and OPTIONS statements position the Comment line. The OPTIONS statement positions the Error line. If the displayed text exceeds the size of the current 4GL window, then 4GL truncates the text to fit the available space.

Sending Output to a Screen Form

You can use the TO clause or the BY NAME clause to display output in the fields of a screen form, using the Formatted mode of display.

Case III: (display output in a screen form)



variable is the name of a variable that is also the name of a field.

Here you cannot use the COLUMN operator nor the AT keyword to position output, since the locations of fields within the form (page 5-4) are fixed.

If 4GL was in Line mode, this form of the DISPLAY statement first clears the screen before sending output to the fields of the form.

Character representations of values are displayed according to data type:

Type of Value	Display
number	Right-justified. If the number does not fit in the field, **4GL** fills the field with asterisks (*) to indicate an overflow.
literal character string, TEXT	Left-justified. If a character string does not fit in the field, **4GL** truncates the display of the value.

BYTE	The field displays the message <byte value>, but actual BYTE values do not appear in the field. (The PROGRAM attribute, as described on page 5-50, can display BYTE and TEXT values.)

Field attributes can change some of these default formats. For example, the LEFT attribute (page 5-31) left-justifies numbers, and the FORMAT attribute can format DATE, DECIMAL, FLOAT, and SMALLFLOAT values. See also the PICTURE attribute (page 5-48) and the USING operator (page 4-91).

The BY NAME Clause

If the variables to be displayed have the same name as screen fields, you can use the BY NAME clause. The BY NAME clause binds the fields to variables implicitly. To use this clause, you must define variables with the same name as the screen fields where they will be displayed. 4GL ignores any record name prefix when matching the names. The names must be unique and unambiguous. If not, this option results in an error, and 4GL sets **status** < 0.

For example, the following statement displays the values for the specified variables in the screen fields with corresponding names (**company**, **address1**, **address2**, **city**, **state**, and **zipcode**):

```
DISPLAY BY NAME p_customer.company, p_customer.address1,
    p_customer.address2, p_customer.city, p_customer.state,
    p_customer.zipcode
```

You can produce the same result by using the THRU or THROUGH notation when listing the fields of the screen record:

```
DISPLAY BY NAME p_customer.company THRU p_customer.zipcode
```

This BY NAME clause displays data to the screen fields of the default screen records. The default screen records are those having the names of the tables defined in the TABLES section of the form specification file. To use a screen array, you define a screen array in addition to the default screen record. This default screen record holds only the first line of the screen array.

For example, the following DISPLAY statement displays the **ordno** variable only in the first line of the screen array (the default screen record):

```
DISPLAY BY NAME p_stock[1].ordno
```

To display **ordno** in all elements of the array, you can use the DISPLAY ARRAY statement, or DISPLAY and the TO clause, as in the next example:

```
FOR i = 1 TO 10
    DISPLAY p_stock[i].ordno TO sc.stock[i].ordno
    ...
END FOR
```

The TO Clause

If the variables do not have the same names as the screen fields, the BY NAME clause is not valid. Instead, you must use the TO clause to map variables to fields explicitly. You can list the fields individually, or you can use the *screen record*.* or *screen record*[*n*].* notation, where *screen record*[*n*].* specifies all the fields in line *n* of a screen array.

You can use the SCROLL statement to move such values up or down, but the DISPLAY ARRAY statement is generally more convenient to use with screen arrays. In the following example, the values in the **p_items** program record are displayed in the first row of the **s_items** screen array:

```
DISPLAY p_items.* TO s_items[1].*
```

The expanded list of screen fields must correspond in order and in number to the expanded list of identifiers after the DISPLAY keyword. Identifiers and their corresponding fields must have the same or compatible data types. For example, the next DISPLAY statement displays the values in the **p_customer** program record in fields of the **s_customer** screen record:

```
DISPLAY p_customer.* TO s_customer.*
```

For this example, the **p_customer** program record and the **s_customer** screen record require compatible declarations. The following DEFINE statement declares the **p_customer** program record:

```
DEFINE      p_customer RECORD
            customer_num LIKE customer.customer_num,
            fname        LIKE customer.fname,
            lname        LIKE customer.lname,
            phone        LIKE customer.phone
            END RECORD
```

This fragment of a form specification declares the **s_customer** screen record:

```
ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.phone;
END

INSTRUCTIONS
SCREEN RECORD s_customer (customer.customer_num,
                          customer.fname,
                          customer.lname,
                          customer.phone)
END
```

In a DISPLAY TO statement, any screen attributes specified in the ATTRIBUTE clause apply to all the fields that you specify after the TO keyword.

The ATTRIBUTE Clause

For general information and the syntax of the ATTRIBUTE clause, see page 3-290. This section describes specific information about using the ATTRIBUTE clause within the DISPLAY statement.

You can use the ATTRIBUTE clause only if you also use the BY NAME, TO, or AT clause. The ATTRIBUTE clause temporarily overrides any default display attributes, or any attributes specified in the OPTIONS or OPEN WINDOW statements for the fields. When the DISPLAY statement completes execution, the default display attributes are restored.

The following DISPLAY statement specifies the attributes REVERSE and BLUE for the message that will be displayed on line 12, starting in the first column:

```
DISPLAY " There are ", num USING "#####",
    " items in the list" AT 12,1
    ATTRIBUTE(REVERSE, BLUE)
```

While the DISPLAY statement is executing, 4GL ignores the INVISIBLE attribute, regardless of whether you specify it in the ATTRIBUTE clause.

Displaying Numeric and Monetary Values

The DBFORMAT and DBMONEY environment variables affect the display of numeric and monetary values as follows. **4GL** will:

- Display the leading currency symbol (as set by DBFORMAT or DBMONEY) for MONEY values. If the FORMAT attribute specifies a leading currency symbol for other data types, then 4GL displays that symbol.
- Omit the thousands separators, unless they are specified by a FORMAT attribute or by the USING operator.
- Display the decimal separator, except for INT or SMALLINT values.
- Display the trailing currency symbol (as set by DBFORMAT or DBMONEY) for MONEY values, unless you specify a FORMAT attribute or the USING operator. In this case, 4GL ignores the trailing currency symbol; the user cannot enter a trailing currency symbol, and 4GL does not display it.

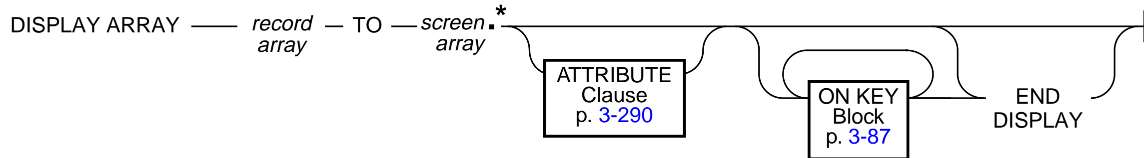
For complete information on DBFORMAT and DBMONEY, refer to Appendix D.

References

INPUT, DISPLAY ARRAY, DISPLAY FORM, OPEN WINDOW, OPTIONS, PRINT

DISPLAY ARRAY

The DISPLAY ARRAY statement displays program array values in a screen array, so that the user can scroll through the screen array.



record array is the identifier of a program array of RECORD variables.

screen array is the identifier of a screen array (page 5-66).

Usage

To use the DISPLAY ARRAY statement, you must do the following:

1. Define the screen array in the form specification file.
2. Use DEFINE to declare an array of program records, whose members correspond in name, data type, and order to the screen array fields.
3. Open and display the screen form with either of the following:
   - The OPEN FORM and DISPLAY FORM statements.
   - The OPEN WINDOW statement with the WITH FORM clause.
4. Fill the program array with data to be displayed, counting the number of program records being filled with retrieved data.
5. Call the SET_COUNT(**x**) function, with **x** the number of filled records.
6. Use the DISPLAY ARRAY statement to display the program array values in the screen array fields.

The SET_COUNT( ) function sets the initial value of the ARR_COUNT( ) function. If you do not call SET_COUNT( ), then 4GL cannot determine how much data to display, and so the screen array remains empty. For a description of the syntax of the built-in SET_COUNT( ) function, see page 4-80.

The DISPLAY ARRAY statement binds the screen array fields to the member records of the program array. The number of *variables* in each record of the program array must be the same as the number of *fields* in each screen record (that is, in a single row of the screen array). Each mapped variable must have the same data type or a compatible data type as the corresponding field.

The size of the *screen array* (from the form specification file) determines the number of program records that 4GL displays at one time on the screen. The size of the *program array* determines how many retrieved rows of data the program can store. The size of the program array can exceed the size of the screen array. In this case, the user can scroll through the rows on the form.

When 4GL encounters a DISPLAY ARRAY statement, it does the following:

1. Displays the program array values in the screen array fields.
2. Moves the cursor to the first field in the first screen record.
3. Waits for the user to press a scroll key (by default, F3 or Page Down to scroll forward, or F4 or Page Up to scroll backwards) or the Accept key (the ESCAPE key by default).

Since the DISPLAY ARRAY statement does not terminate until the user presses the Accept or Interrupt key, you may want to display a message informing the user. By default, 4GL displays variables and constants as follows:

- Right-justifies number values in a screen field.
- Left-justifies character values in a screen field.
- Truncates the displayed value, if a character value is longer than the field.
- Fills the field with asterisks (*) to indicate an overflow, if a number value is larger than the field can display.
- If the field contains a BYTE value, displays `<byte value>` in the field.

The following are among the topics that are described in this section:

Topic	Page
The ATTRIBUTE Clause	3-86
The ON KEY Blocks	3-87
The EXIT DISPLAY Statement	3-89
The END DISPLAY Keywords	3-89
Using Built-In Functions and Operators	3-90
Scrolling During the DISPLAY ARRAY Statement	3-91
Completing the DISPLAY ARRAY Statement	3-92

The ATTRIBUTE Clause

For general information and the syntax of the ATTRIBUTE clause, see page 3-290. This section describes specific information about using the ATTRIBUTE clause within a DISPLAY ARRAY statement.

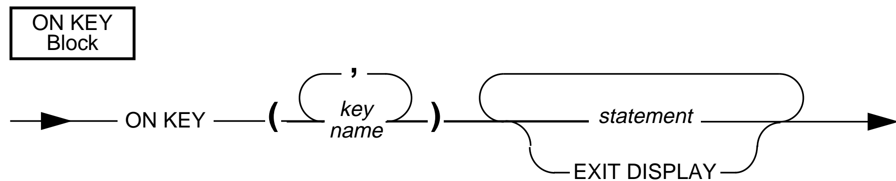
The attributes that you include apply to all of the fields in *screen array*. For example, the following DISPLAY ARRAY statement displays items in RED:

```
DISPLAY ARRAY p_items TO s_items.* ATTRIBUTE (RED)
```

The ATTRIBUTE clause specifications overrides all default attributes, and temporarily override any display attributes that the OPTIONS or the OPEN WINDOW statement specified for these fields. While the DISPLAY ARRAY statement is executing, 4GL ignores the INVISIBLE attribute.

The ON KEY Blocks

The ON KEY keywords specify a block of statements to be executed when the user selects one of the specified keys. This is the syntax of the ON KEY block:



key name is one or more of these keywords, in uppercase or lowercase letters, separated by commas, to specify a key:

ACCEPT	HELP	NEXT *or*	RETURN
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS *or*	TAB
ESC *or* ESCAPE	LEFT	PREVPAGE	UP
F1 *through* F64			
CONTROL-*char*	(except A, D, H, I, J, L, M, R, or X)		

statement is an SQL statement or some other 4GL statement.

For *key name* you can also substitute the NEXTPAGE keyword as a synonym for NEXT, and PREVPAGE as a synonym for PREVIOUS.

4GL executes the *statements* specified in the ON KEY block when the user presses one of the keys that you specify. 4GL deactivates the form while executing statements in an ON KEY block. After executing the statements, 4GL reactivates the form, allowing the user to continue viewing the fields.

You can enter uppercase or lowercase *key* specifications. Some keys require special consideration before you reference them in an ON KEY clause:

Key	Special Considerations
ESC or ESCAPE	Specify another key as the Accept key in the OPTIONS statement, because ESC is the default Accept key.
INTERRUPT	You must execute a DEFER INTERRUPT statement, so that when the user presses the Interrupt key, **4GL** executes the statements in the ON KEY clause and sets **int_flag** to nonzero for the current task, but does not terminate the DISPLAY ARRAY statement. **4GL** also executes the statements in this ON KEY clause if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, **4GL** sets **quit_flag** to non-zero for the current task.
F3	Specify another key as the Next Page key in the OPTIONS statement, because F3 is the default Next key.
F4	Specify another key as the Previous Page key in the OPTIONS statement, because F4 is the default Previous key.
CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these keys for field editing.
I, J, and M	If you specify these keys in the ON KEY clause, the key is "trapped" by **4GL** to activate the ON KEY clause. The standard effect of these keys (TAB, LINEFEED, and RETURN, respectively) is not available to the user. For example, if CONTROL-M appears in an ON KEY clause, the user cannot press RETURN to advance the cursor to the next field.

You may not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

After executing the statements in the ON KEY block, 4GL resumes the display with the cursor in the same location as before the ON KEY block, unless it encounters EXIT DISPLAY within the block. (In this case, program execution resumes at the statement following the DISPLAY ARRAY statement.)

The following ON KEY clause specifies two keys to display a Help message:

```
ON KEY (f1, control-w) CALL customer_help()
```

The EXIT DISPLAY Statement

The EXIT DISPLAY statement terminates the DISPLAY ARRAY statement. When it encounters an EXIT DISPLAY statement, 4GL does the following:

1. Skips all subsequent statements between the EXIT DISPLAY keywords and the END DISPLAY keywords.
2. Resumes execution at the statement after the END DISPLAY keywords.

For example, the EXIT DISPLAY statement terminates the following DISPLAY ARRAY statement if the user presses F5 and the value of **amt_received** in the current program array record is greater than 1000:

```
DISPLAY ARRAY p_receipts TO s_receipts.*
    ON KEY (F5)
        LET x = arr_curr()
        IF p_receipts[x].amt_received > 1000 THEN
            CALL get_allocation(p_receipts[x].receipt_num)
            EXIT DISPLAY
        END IF
END DISPLAY
```

The END DISPLAY Keywords

The END DISPLAY keywords terminate the DISPLAY ARRAY statement. Each of these conditions requires that you include the END DISPLAY keywords:

- The DISPLAY ARRAY statement includes one or more ON KEY blocks.
- The DISPLAY ARRAY statement is specified in a form management block of a CONSTRUCT, INPUT, or INPUT ARRAY statement, and an ON KEY block of the enclosing statement follows the DISPLAY ARRAY statement.
- The DISPLAY ARRAY statement is specified within an ON KEY block in a PROMPT statement or in another DISPLAY ARRAY statement.

The following DISPLAY ARRAY statement must include the END DISPLAY keywords because it immediately precedes an ON KEY block that belongs to an INPUT statement:

```
INPUT BY NAME p_customer.*
    AFTER FIELD company
        ...
        DISPLAY ARRAY pa_array TO sc_array.*
        END DISPLAY
    ON KEY (CONTROL_B)
        ...
END INPUT
```

Otherwise, it would be ambiguous whether the ON KEY block were part of the INPUT statement or part of the DISPLAY ARRAY statement.

Here the END DISPLAY keywords are required because of the ON KEY clause:

```
DISPLAY ARRAY p_items TO s_items.*ON KEY (CONTROL_W)
        CALL get_help()
END DISPLAY
```

Using Built-In Functions and Operators

INFORMIX-4GL provides several built-in functions to use in a DISPLAY ARRAY statement. These are described in Chapter 4 and are summarized here.

You can use the following built-in functions to keep track of the relative states of the screen cursor, the program array, and the screen array:

Function	**Description**
ARR_CURR( )	Returns the number of the *current record* of the program array. This corresponds to the position of the screen cursor at the beginning of the ON KEY control block, not the line to which the screen cursor moves after execution of the block.
ARR_COUNT( )	Returns the current number of records in the program array.
SCR_LINE( )	Returns the number of the current line within the screen array. This number can be different from the value returned by ARR_CURR( ) if the program array is larger than the screen array.
SET_COUNT( )	Takes the number of rows currently in the program array as an argument, and sets the initial value of ARR_COUNT( ).

DISPLAY ARRAY also supports the following built-in functions and operators that allow you to access field buffers and keystroke buffers:

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has "touched" (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or Arrow keys) does not mark a field as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form.
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user while in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is passed as its operand is the name of the current field.

For more about these built-in 4GL functions and operators, see Chapter 4. Each field in a form has only one field buffer, and a buffer cannot be used by two different statements simultaneously. If you plan to display more than once the same form with data entry fields, you can open a new 4GL window, and then open and display in it a second copy of the form. INFORMIX-4GL allocates a separate set of buffers to each form, and you can be certain that your program is retrieving the correct field values.

Scrolling During the DISPLAY ARRAY Statement

Users can select these keys to scroll through the screen array:

Key	Effect
[↓], [→]	Moves the cursor down one row at a time. If the cursor was on the last row of the screen array before the user pressed one of these arrow keys, 4GL scrolls the program array data up one row. If the last row in the program array is already in the last row of the screen array, pressing one of these keys generates a message that says there are no more rows in that direction.
[↑], [←]	Moves the cursor up one row at a time. If the cursor was on the first row of the screen array before the user pressed one of these arrow keys, 4GL scrolls the program array data down one row. If the first row in the program array is already in the first row of the screen array, pressing one of these keys generates a message that says there are no more rows in that direction.
F3	Scrolls the display to the next full page of program array records. You can reset this key by using the NEXT KEY option of the OPTIONS statement.
F4	Scrolls the display to the previous full page of program array records. You can reset this key by using the PREVIOUS KEY option of the OPTIONS statement.

Completing the DISPLAY ARRAY Statement

The following conditions terminate the DISPLAY ARRAY statement:

- The user chooses any of the following keys:
  - Accept
  - Interrupt
  - Quit
- 4GL executes the EXIT DISPLAY statement

By default, the Accept, Interrupt, or Quit keys terminate the DISPLAY ARRAY statement. Each of these actions also deactivates the form. (But pressing the Interrupt or Quit key can immediately terminate the program, unless the program also includes the DEFER INTERRUPT and DEFER QUIT statements.)

If 4GL previously executed a DEFER INTERRUPT statement in the program, the Interrupt key causes 4GL to do the following:

- Set the global variable **int_flag** to a nonzero value.
- Terminate the DISPLAY ARRAY statement, but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, pressing the Quit key causes 4GL to do the following:

- Set the global variable **quit_flag** to a nonzero value.
- Terminate the DISPLAY ARRAY statement, but not the 4GL program.

The following program fragment displays a program array **p_customer** in the fields of a screen array called **s_customer**:

```
OPEN FORM f_customer FROM "f_customer"
DISPLAY FORM f_customer
...
DECLARE c_custs CURSOR FOR
    SELECT customer_num, company
    FROM customer
    WHERE state = "CA"
LET counter = 1
FOREACH c_custs INTO p_customers[counter].*
    LET counter = counter + 1
END FOREACH
...
CALL SET_COUNT(counter - 1)
DISPLAY ARRAY p_customers TO s_customers.*
```

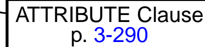
References

ATTRIBUTE, DISPLAY, INPUT ARRAY, OPEN WINDOW, OPTIONS, SCROLL

DISPLAY FORM

The DISPLAY FORM statement displays a compiled 4GL screen form.

DISPLAY FORM *form*



form is the identifier of a 4GL screen form.

Usage

Before you can use a compiled form to display information on the screen or to accept keyboard input from the user, you must do the following:

- First, use the OPEN FORM statement to declare the name of the form.
- Then use the DISPLAY FORM statement to display the form on the screen.

The *form* name specifies the name of the screen form to be displayed. The DISPLAY FORM statement is not required if you open and display a form by using the WITH FORM option of the OPEN WINDOW statement (page 3-221).

An error occurs if the current 4GL window is too small to display the form.

Form Attributes

The DISPLAY FORM statement ignores the INVISIBLE attribute. 4GL applies any other display attributes that you specify in the ATTRIBUTE clause to any fields that have not been assigned attributes by the ATTRIBUTES section of the form specification file, or by the **syscolatt** table, or by the OPTIONS statement. If the form is displayed in a 4GL window, color attributes from the DISPLAY FORM statement supersede any from the OPEN WINDOW statement.

If subsequent CONSTRUCT, DISPLAY, or DISPLAY ARRAY statements that include an ATTRIBUTE clause reference the form, however, their attributes take precedence over those of the DISPLAY FORM statement. (For information on display attributes and the order of precedence among conflicting attribute specifications, see the section on ATTRIBUTE Clauses on page 3-290.

Reserved Lines

DISPLAY FORM displays the specified form in the current 4GL window, or in the **4GL** screen itself, if no other 4GLwindow is open. The form begins in the line that was indicated by the FORM LINE specification of the OPEN WINDOW or OPTIONS statements. This specification positions the first

line of the form relative to the top of the current 4GL window. If you provided no FORM LINE specification, the default Form line is 3. On a default screen display, the reserved lines are positioned as follows:

Default Location	Reserved for
First line	Prompt line (output from PROMPT statement); *also* Menu line (*command value* from MENU statement).
Second line	Message line (output from MESSAGE statement; *also* the *description value* output from MENU statement).
Third line	Form line (output from DISPLAY FORM statement).
Second-to-last line	Comment line (output from COMMENT attribute) when SCREEN is the current **4GL** window.
Last line	Error line (output from ERROR statement); *also* Comment line in any **4GL** window except SCREEN.

For example, the following statements display the **cust_form** form in the **4GL** screen (or in the current 4GL window):

```
OPEN FORM cust_form FROM "customer"
DISPLAY FORM cust_form
```

The OPTIONS statement can change the default position of all the reserved lines, including that of the Form line, for all 4GL windows, including the entire 4GL screen (specified as SCREEN). You can also reposition the Form line for a specific 4GL window only, by using an ATTRIBUTE clause in the OPEN WINDOW statement.

The following statements make line 6 the Form line for all 4GL windows, and then displays **cust_form**:

```
OPTIONS FORM LINE 6
OPEN FORM cust_for FROM "customer"
DISPLAY FORM cust_form
```

References

CLEAR, CLOSE FORM, OPEN FORM, OPEN WINDOW, OPTIONS

END

The END keyword marks the end of a compound 4GL statement.

```
END ──────────────── keyword ──────────────────────┤
```

keyword is a keyword that specifies the name of the 4GL statement to be delimited, from among the keywords listed below.

Usage

The END keyword marks the last line of a compound 4GL statement. This is a compile-time indicator of the end of the *statement* construct. (Use the EXIT keyword, rather than END, to terminate *execution* of a compound statement.)

Several compound statements of 4GL support END keywords to mark the end of the *statement* construct within the source module:

CASE	FOREACH	INPUT	PROMPT
CONSTRUCT	FUNCTION	INPUT ARRAY	REPORT
DISPLAY ARRAY	GLOBALS	MAIN	WHILE
FOR	IF	MENU	

The END DISPLAY keyword delimits the DISPLAY ARRAY statement, and END INPUT delimits both INPUT and INPUT ARRAY. Unlike EXIT *statement* clauses (page 3-98), no more than one END *statement* clause can appear within the specified *statement*, but most compound statements of 4GL can be nested.

This statement fragment uses END MENU to delimit a MENU statement:

```
MENU "MAIN"
    . . .
END MENU
```

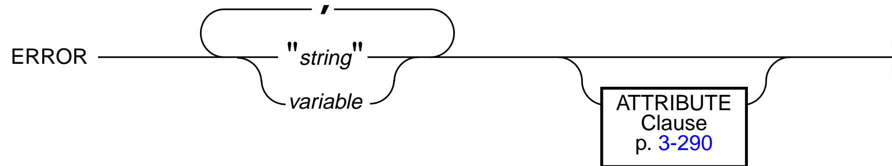
The END keyword can also delimit RECORD declarations in DEFINE statements, as well as sections of form specification files.

References

CASE, DISPLAY ARRAY, FOR, FOREACH, FUNCTION, GLOBALS, IF, INPUT, INPUT ARRAY, MAIN, MENU, PROMPT, REPORT, WHILE

ERROR

The ERROR statement displays an error message on the Error line and rings the terminal bell.



string is a quoted string of a length no greater than the number of characters than the Error line of the current 4GL window can display.

variable is the name of a CHAR or VARCHAR variable whose contents are to be displayed on the Error line of the **4GL** screen.

Usage

The *string* or *variable* specifies all or part of the text of a screen message to be displayed on the Error line.

You can specify any combination of character variables and literal character strings for the message. 4GL generates the message to display by replacing any variables with their values, and concatenating the returned strings. The total length of this message must not be greater than the number of characters that the Error line can display in a single line of the **4GL** screen. The message text remains on the screen until the user presses the next key.

The error text appears in a borderless single-line 4GL window on the Error line. This 4GL window opens to display your text when ERROR is executed, and closes at the next keystroke by the user. When this 4GL window closes, any underlying display on the same line becomes visible again.

The position of the Error line is determined by the most recently executed ERROR LINE specification in the OPTIONS statement. Otherwise, the default Error line position is the last line of the screen. Because the Error line is positioned relative to the *screen*, rather than to the *current window*, you cannot use the OPEN WINDOW statement to reposition the Error line.

You can use the CLIPPED and USING operators in the ERROR statement, as illustrated in the following examples:

```
ERROR p_orders.order_num USING "#####", " is not valid."

ERROR pattern CLIPPED, " has no match."
```

You can also use the ASCII and COLUMN operators, and other features of 4GL character expressions. (For more information on the built-in functions and operators of 4GL, see Chapter 4.)

The ATTRIBUTE Clause

The ATTRIBUTE clause options are described in the ATTRIBUTE Clause section on page 3-290. The default display attribute for the Error line is REVERSE. You can use the ATTRIBUTE clause to specify some other attribute. 4GL ignores the INVISIBLE attribute if you include it with another attribute in the ATTRIBUTE clause of the ERROR statement. If the INVISIBLE attribute is the only attribute that you specify, 4GL displays the ERROR text as NORMAL.

The next example specifies BLUE and BLINK attributes for the ERROR text:

```
ERROR “Unable to insert items” ATTRIBUTE(BLUE, BLINK)
```

System Error Messages

The Error line also displays *system error messages*. These can provide you with useful diagnostic information while you are developing 4GL programs, but they may not be helpful to users of your application.

One way to avoid displaying system error messages is to use the WHENEVER statement to trap run-time errors. The WHENEVER statement can call a function that executes an ERROR statement, displaying a screen message that is more suitable for your users.

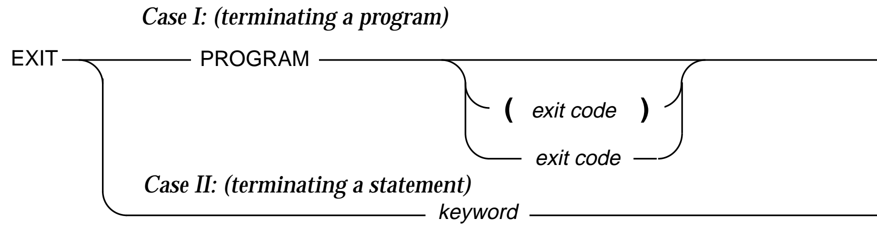
Some run-time errors cannot be trapped by the WHENEVER statement. The description of “Exception Handling” on page 2-23 includes a list of error messages that are currently untrappable.

References

DISPLAY, MESSAGE, OPTIONS, PROMPT, WHENEVER

EXIT

The EXIT statement transfers control out of a control structure: a block, a loop, a CASE statement, an interface statement, or out of the program itself.



exit code is an integer expression (page 3-338).

keyword is a keyword that specifies the current statement from which control of execution is to be transferred, from among those in the list that appears below on this page.

Usage

The EXIT PROGRAM statement terminates the program that is currently executing. Other forms of EXIT transfer control from the current control structure to whatever statement follows the corresponding END *keyword* keywords.

Leaving a Control Structure

Some compound statements support EXIT *statement* to terminate execution of the current *statement* and pass control of execution to the next statement:

EXIT CASE	EXIT FOR	EXIT MENU
EXIT CONSTRUCT	EXIT FOREACH	EXIT WHILE
EXIT DISPLAY	EXIT INPUT	

Here EXIT DISPLAY exits from DISPLAY ARRAY (but not DISPLAY) statements, and EXIT INPUT can exit from both INPUT ARRAY and INPUT statements.

Unlike EXIT PROGRAM, these other EXIT statements can only appear within the specified *statement.* For example, EXIT FOR can occur only in a FOR loop; if it is executed, it transfers control to the statement following the END FOR keywords that mark the end of that FOR statement.

Similarly, EXIT MENU can appear only within a control block of a MENU statement; it transfers control to the statement that follows the corresponding END MENU keywords of the same MENU statement.

Leaving the Program

The EXIT PROGRAM statement terminates execution of the 4GL program. After 4GL encounters the EXIT PROGRAM statement anywhere within the program, no subsequent statements are executed, and control returns to the operating system (or to whatever process invoked the 4GL program). For example, here EXIT PROGRAM appears in a MENU statement:

```
MENU "MAIN"
    ...
    COMMAND "Quit" "Exit from the program"
        CLEAR SCREEN
        EXIT PROGRAM
END MENU
```

If 4GL encounters the END MAIN keywords in the MAIN block, END MAIN terminates the program, as if you specified EXIT PROGRAM (0). If you are using the INFORMIX-4GL Interactive Debugger, a program that EXIT PROGRAM terminates can be examined subsequently by the WHERE or STACK commands of the Debugger, as if an abnormal termination had occurred.

The Exit Code Value

The *exit code* returns the status code when a process terminates. The status code is a whole-number value, usually less than 256. The RETURNING clause of the RUN statement instructs 4GL to save the *exit code* from the EXIT PROGRAM statement, if RUN invokes a 4GL program that EXIT PROGRAM terminates. When the 4GL program that RUN specifies completes execution, RUN can return an integer variable that contains two bytes of termination status information:

- The low byte contains the termination status of whatever RUN executes. You can recover this by calculating the value of (*integer value* modulo 256).
- The high byte contains the low byte from the EXIT PROGRAM statement of the 4GL program that RUN executes. You can recover this returned code by dividing *integer value* by 256.

See the RUN statement (page 3-266) for an example of using RUN and EXIT PROGRAM to examine termination status and exit codes from 4GL programs that RUN invoked and EXIT PROGRAM terminated.

References

CONTINUE, END, GOTO, LABEL, MAIN, RUN

FINISH REPORT

The FINISH REPORT statement completes processing of a 4GL report.

```
FINISH REPORT   report ─────────────────────────────────────┤
```

report is the name of a 4GL report, as declared in the REPORT statement.

Usage

You can use the FINISH REPORT statement to indicate the end of a ***report driver*** (page 6-5) and to complete processing of the report. FINISH REPORT must follow a START REPORT statement, and at least one OUTPUT TO REPORT statement, that reference the same report.

If the REPORT definition includes an ORDER BY section with no EXTERNAL keyword (page 6-22), or specifies aggregates (page 6-46) based on all the input records, 4GL makes two passes through the input records. During the first pass, it uses the database engine to sort the data, and then stores the sorted values in a temporary file. During the second pass, it calculates any aggregate values, and produces output from data in the temporary files.

Executing the FINISH REPORT statement performs the following actions:

- Completes the second pass, if *report* is a two-pass report. These "second pass" activities handle the calculation and output of any aggregate values (page 6-46) that are based on all the input records in the report, such as COUNT(*) or PERCENT(*) with no GROUP qualifier.
- Executes any AFTER GROUP OF control blocks (page 6-29).
- Executes any PAGE HEADER (page 6-37), ON LAST ROW (page 6-36), and PAGE TRAILER control blocks (page 6-38) needed to complete the report.
- Copies any data from the output buffers of the report to what START REPORT or the report definition (page 6-13) specified as the destination. If none was specified, output goes to the Report window (page 6-14).
- Closes the Select cursor on any temporary table that was created to order the input records or to perform aggregate calculations.
- Deallocates memory for any local BYTE or TEXT variables in the report.
- Terminates processing of the 4GL report, and deletes from the database any files that held temporary tables for a two-pass report.

The following program creates a report based on data in the **orders** table:

```
DATABASE stores2
MAIN
    DEFINE p_orders RECORD LIKE orders.*
    DECLARE q_ordcurs CURSOR FOR SELECT * FROM orders
    START REPORT ord_list TO "ord_listing"
    FOREACH q_ordcurs INTO p_orders
        OUTPUT TO REPORT ord_list(p_orders)
    END FOREACH
    FINISH REPORT ord_list
END MAIN
REPORT ord_list(r_orders)
    DEFINE r_orders RECORD LIKE orders.*
    FORMAT EVERY ROW
END REPORT
```

Temporary Tables Created by Reports

The temporary tables that 4GL reports use for sorting input records or for calculating aggregates in two-pass reports are stored in the current database. If you do not open any database, or if the CLOSE DATABASE statement closes the current database, then a run-time error occurs when 4GL cannot create or access the temporary table that is required for a two-pass report.

Similarly, the FINISH REPORT statement cannot access temporary tables in more than one database. An error can occur if the DATABASE statement opens a different database while a two-pass 4GL report is being processed. The following program fragment, for example, produces a run-time error if the **produce** report requires two passes:

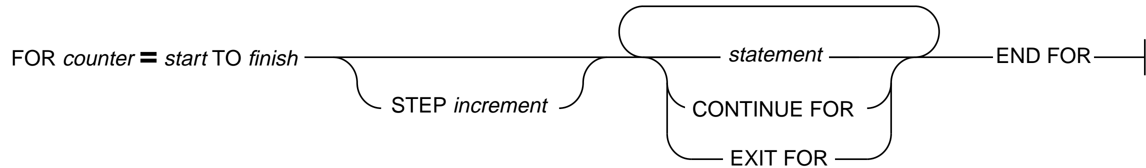
```
DATABASE apples
  . . .
START REPORT produce                   --database is apples
  . . .
OUTPUT TO REPORT produce(input_rex)
  . . .
DATABASE oranges                   --new database is oranges
FINISH REPORT produce --cannot access files in apples database
```

References

DATABASE, OUTPUT TO REPORT, REPORT, START REPORT

FOR

The FOR statement executes a statement block a specified number of times.



counter is a variable of type INTEGER or SMALLINT that serves as an index for the *statement* block.

finish is an integer expression (page 3-338) to specify an upper limit for *counter.*

increment is an integer expression (page 3-338) whose value is added to *counter* after each iteration of the *statement* block.

start is an integer expression (page 3-338) to set an initial *counter* value.

statement is an SQL statement or other 4GL statement. (This *statement* block is sometimes called "the FOR loop.")

Usage

The FOR statement executes the statements up to the END FOR statement a specified number of times, or until EXIT FOR terminates the FOR statement. (Use the WHILE statement, rather than FOR, if you cannot specify an upper limit on how many times the program needs to repeat a statement block, but you can specify a Boolean condition for leaving the block.)

The TO Clause

On each iteration through the *statements*, the *counter* is set to a different value. On the first pass through the loop, the counter is set to the initial *expression* at the left of the TO keyword. Thereafter, the value of the *increment* expression in the STEP clause specification (or by default, 1) is added to *counter* in each pass through the block of *statements.*

When the sign of the difference between the values of *counter* and the *finish* expression at the right of the TO keyword changes, 4GL exits from the FOR loop. Execution resumes at the statement following the END FOR keywords.

For example, this statement clears four records of the **s_items** screen array:

```
FOR counter = 1 TO 4
    CLEAR s_items[counter].*
END FOR
```

***Note:** The FOR loop terminates after the iteration for which the left- and right- hand expressions are equal. If either expression returns NULL, the loop cannot terminate, because in that case the Boolean expression "left = right" cannot become TRUE.*

The STEP Clause

Use the STEP clause to tell 4GL the number by which to increment the counter. For example, this FOR statement increments the counter by 2:

```
FOR idx = 1 TO 12 STEP 2
    DISPLAY month_names[idx] TO sc_month[i]
    LET i = i + 1
END FOR
```

If you use a negative STEP value, specify the second *expression* in the TO clause as smaller than the first value in the range.

Before processing the block of statements, 4GL first tests the counter value against the terminating value. For example, if the STEP value is positive and the counter value is greater than the last value in the range, 4GL skips over the statements in the loop without executing them.

The CONTINUE FOR Statement

Use the CONTINUE FOR statement to interrupt the current iteration and start the next iteration of the statement block. To execute a CONTINUE FOR statement, 4GL does the following:

- Skips the remaining statements between the CONTINUE FOR and END FOR keywords.
- Increments the counter variable and tests it.
- If the counter does not exceed the final value, then 4GL goes back to the beginning of the loop and performs another iteration. Otherwise, 4GL continues execution after the END FOR keywords.

The EXIT FOR Statement

Use the EXIT FOR statement to terminate the statement block. When 4GL encounters this statement, it: skips any statements between the EXIT FOR and END FOR keywords. Execution resumes at the first statement immediately after the END FOR keywords.

The END FOR Keywords

Use this to indicate the end of the FOR loop. Upon encountering the END FOR keywords, 4GL increments the counter and compares it with the expression that immediately follows the TO keyword. If the counter exceeds this value, then 4GL terminates the FOR loop and executes the statement following the END FOR keywords.

Databases with Transactions

If your database has transaction logging, and the FOR loop includes one or more SQL statements that modify the database, then it is advisable that the entire FOR loop be within a transaction. Otherwise, if an error occurs after some of the SQL statements within the FOR loop have executed, but before the loop has terminated, the user may face two potential problems:

- It may be difficult to determine the extent to which the integrity of the database has been compromised; and
- If the database has been corrupted, it may be difficult to restore it to its condition prior to the execution of the FOR loop.

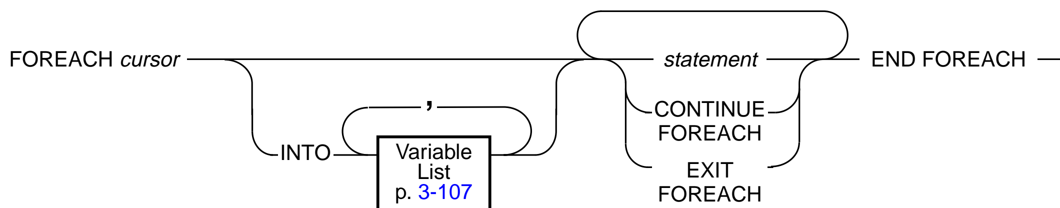
The same data integrity considerations also apply to FOREACH and WHILE loops that include SQL statements in 4GL programs. (See the *Informix Guide to SQL: Tutorial* for more information about the SQL concepts and statements that support data integrity through transactions.)

References

CONTINUE, FOREACH, WHILE

FOREACH

The FOREACH statement applies a series of actions to each row of data that is returned from a query by a cursor.



cursor is the name of a previously declared SQL cursor.

statement is an SQL statement or other 4GL statement.

Usage

Use the FOREACH statement to retrieve and process rows selected by a query. The FOREACH statement is equivalent to using the OPEN, FETCH and CLOSE statements. The FOREACH statement has these effects:

- Opens the specified cursor
- Fetches the rows selected
- And then closes the cursor

You must declare the cursor (by using the DECLARE statement) before the FOREACH statement can retrieve the rows. A compile-time error occurs unless the cursor was declared prior to this point in the source module. You can reference a sequential cursor, a SCROLL cursor, a cursor WITH HOLD, or FOR UPDATE, but FOREACH only processes rows in sequential order.

FOREACH does not have the equivalent of the USING clause that the OPEN statement of SQL supports. The FOREACH statement can only open cursors for SELECT statements that do not contain unknown parameters.

The FOREACH statement performs successive fetches until all rows specified by the SELECT statement are retrieved. Then the cursor is automatically closed. It is also closed if a WHENEVER NOT FOUND statement within the FOREACH loop detects a NOTFOUND condition (that is, `status` = 100).

Implicit FETCH statements that FOREACH executes with an Update cursor can support promotable locks. (See the *Informix Guide to SQL: Reference*.)

The following topics are described in this section:

Topic	Page
Cursor Names	3-106
The INTO Clause	3-107
The FOREACH Statement Block	3-108
The CONTINUE FOREACH Statement	3-108
The EXIT FOREACH Statement	3-109
The END FOREACH Keywords	3-109

Cursor Names

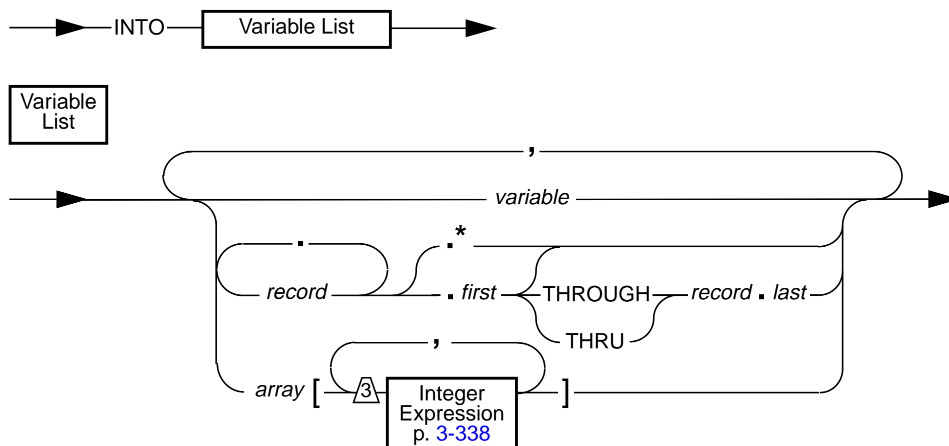
Each SQL cursor has a name. The name of a cursor must be specified in the FOREACH statement. Follow the FOREACH keyword with a *cursor name* that a DECLARE statement declared earlier in the same module. A run-time error can occur if the FOREACH statement does not specify the name of a cursor that a DECLARE statement has previously declared.

The next example fetches values retrieved by the **c_orders** cursor. For each retrieved row, 4GL increments the **counter** variable by 1, invokes a function called **scan( )**, and passes the values of **ord_num**, **cust_num**, and **comp**. If the query does not return any rows, 4GL ignores the FOREACH loop and resumes processing with the statement that immediately follows the END FOREACH keywords. This IF statement examines the **counter** variable, and displays a message on the Error line if the query returned no rows.

```
PROMPT "Enter cut-off date for orders: " FOR o_date
DECLARE c_orders CURSOR FOR
    SELECT order_num, orders.customer.num, company
    INTO ord_num, cust_num, comp FROM orders o, customer c
    WHERE o.customer_num = c.customer_num
        AND order_date < o_date
LET counter = 1
FOREACH c_orders
    LET counter = counter + 1
    CALL scan(ord_num, cust_num, comp)
END FOREACH
IF counter = 0 THEN
    ERROR "No orders before ", o_date
END IF
```

The INTO Clause

The INTO clause specifies a variable or a comma-separated list of variables in which to store values from each row that is returned by the query:



array is the name of a program array.

first is the name of a member variable in which to store a value.

last is another member of *record* that was declared later than *member.*

record is the name of a variable of the RECORD data type.

variable is the name of a simple variable to store the retrieved value.

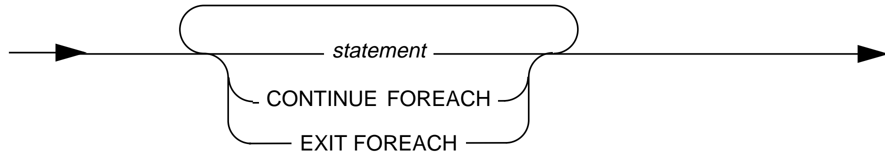
You can include the INTO clause in the SELECT statement associated with the cursor, or in the FOREACH statement, but not in both. To retrieve rows into a program array, however, you must place the INTO clause in the FOREACH statement. For example, the following FOREACH statement stores the retrieved rows in the **p_items** program array:

```
LET counter = 1
FOREACH my_curs INTO p_items[counter].*
    LET counter = counter + 1
    IF counter > 10 THEN
        CALL mess ("Ten or more items.")
        EXIT FOREACH
    END IF
END FOREACH
```

The number and order of variables in the INTO clause must match the number and order of the columns in the active set of rows that are retrieved by the cursor, and must be of compatible data types.

The FOREACH Statement Block

These statements are executed after each row of the active set is fetched:



This block is sometimes called "the FOREACH loop." If the cursor returns no rows, then no statements in this loop are executed, and program control passes to the first statement that follows the END FOREACH keywords.

If the specified cursor is an Update cursor, the statement block can include statements to modify retrieved rows. See the *Informix Guide to SQL: Reference.*

Databases with Transactions

If your database has *transaction logging*, then it is advisable that the entire FOREACH statement block be within a transaction. Otherwise, if an error occurs after some of the SQL statements within the FOREACH statement block have executed, but before the loop has terminated, the user may face two potential problems:

- It may be difficult to determine the extent to which the integrity of the database has been compromised; and
- If the database has been corrupted, it may be difficult to restore it to its condition prior to the execution of the FOREACH loop.

These considerations apply to FOR and WHILE loops that can change the database. (See *Informix Guide to SQL: Tutorial* for information about the SQL concepts and statements that support data integrity through transactions.)

The CONTINUE FOREACH Statement

Use the CONTINUE FOREACH statement to interrupt processing of the current row and start processing the next row. 4GL fetches the next row and resumes processing at the first statement in the FOREACH statement block. For example, if **total_price** is less than 1000 in the next example, 4GL increments **smallOrders**, fetches the next row, and executes the IF statement. If **total_price** is equal to or greater than 1000, 4GL proceeds to the next statement in the FOREACH block, in this case, the OUTPUT TO REPORT statement:

```
LET smallOrders = 1
FOREACH orderC
    IF orderP.total_price < 1000 THEN
        LET smallOrders = smallOrders + 1
        CONTINUE FOREACH
    END IF
    OUTPUT TO REPORT order_list (orderR.*, smallOrders)
    ...
END FOREACH
```

The EXIT FOREACH Statement

Use this statement to interrupt processing and ignore the remaining rows. Upon encountering an EXIT FOREACH statement, 4GL skips the statements between the EXIT FOREACH and the END FOREACH keywords. Execution resumes at the statement following the END FOREACH keywords. For example, if the **status** variable is not equal to 0 in the following program segment, 4GL displays a message and then exits from the FOREACH loop:

```
FOREACH orderC
    ...
    IF status != 0 THEN
        MESSAGE "Error on output to report."
        EXIT FOREACH
    END IF
END FOREACH
```

The END FOREACH Keywords

Use the END FOREACH keywords to indicate the end of the FOREACH loop. When 4GL encounters the END FOREACH keywords, it re-executes the loop until no more rows returned by the query remain. Otherwise, it executes the statement that follows the END FOREACH keywords.

For example, if the **status** variable is not equal to 0 in the following program fragment, 4GL displays a message and exits from the FOREACH loop:

```
DECLARE orderC CURSOR FOR
    SELECT * INTO orderR.* FROM orders
    WHERE order_date BETWEEN start_date AND end_date
START REPORT order_list
LET smallOrders = 0
```

```
FOREACH orderC
    IF orderR.total_price < 1000 THEN
        LET smallOrders = smallOrders + 1
        CONTINUE FOREACH
    END IF
    OUTPUT TO REPORT order_list (orderR.*, smallOrders)
    IF status != 0 THEN
        MESSAGE "Error on output to report."
        EXIT FOREACH
    END IF
END FOREACH
FINISH REPORT order_list
```

The next example creates a cursor **c_query**, based on search criteria entered by the user. For each row retrieved by the SELECT statement defined for the cursor, this example displays the row on the screen and waits for the user to request the next row. If no rows are selected, then 4GL displays a message.

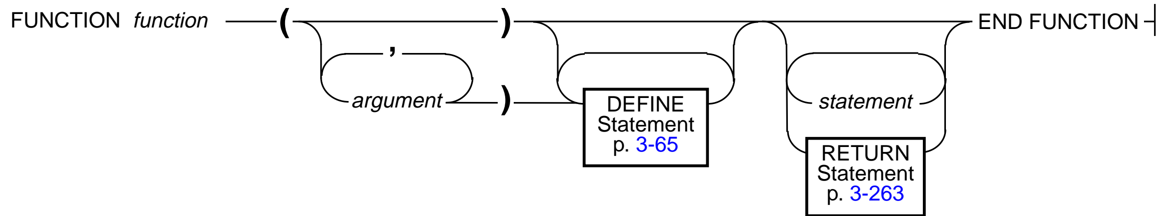
```
DEFINE stmt1, query1 CHAR(300),
        p_customer RECORD LIKE customer.*
CONSTRUCT BY NAME query1 ON customer.*
LET stmt1 = "SELECT * FROM customer ",
    "WHERE ", query1 CLIPPED
PREPARE stmt_1 FROM stmt1
DECLARE c_query CURSOR FOR stmt_1
LET exist = 0
FOREACH c_query INTO p_customer.*
    LET exist = 1
    DISPLAY BY NAME p_customer.*
    PROMPT "Do you want to see the next customer (y/n): "
        FOR ANSWER
    IF answer MATCHES "[Nn]" THEN
        EXIT FOREACH
    END IF
END FOREACH
IF exist = 0 THEN
    MESSAGE "No rows found."
END IF
```

References

CONTINUE, FETCH, FOR, OPEN, WHILE, WHENEVER

FUNCTION

This statement defines a FUNCTION program block.



argument is the name of a formal argument to this function. The DEFINE statement that follows must specify a data type for each argument.

function is the identifier that you declare for this 4GL function.

statement is an statement or other 4GL statement.

Usage

As Chapter 3 explains, a 4GL *function* is a named block of statements. The FUNCTION statement defines a 4GL function that can be invoked from any module of your program. The FUNCTION statement has two effects:

- To *declare* the name of a function and of any formal arguments (page 3-112). 4GL imposes no limit on the number or size of formal arguments.
- To *define* the corresponding FUNCTION program block (page 3-113).

The FUNCTION statement cannot appear within the MAIN statement, nor in a REPORT statement, nor within another FUNCTION statement.

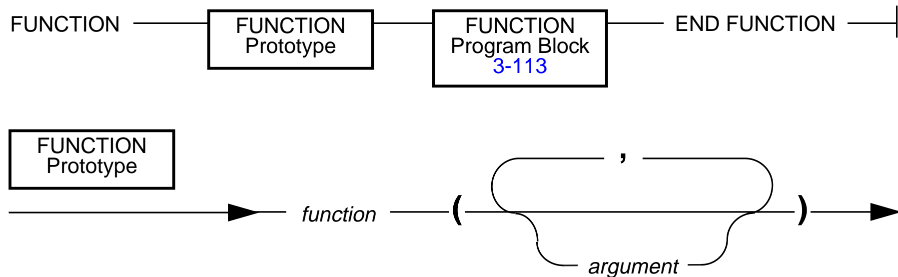
If the function returns a single value, it can be invoked as an operand within a 4GL expression (page 3-332). Otherwise, you must invoke it with the CALL statement (page 3-16). An error results if the list of returned values in the RETURN statement conflicts in number or in data types with the RETURNING clause of the CALL statement that invokes the function (page 3-19).

Topics that are described in this section include the following:

Topic	Page
The Prototype of the Function	3-112
The Argument List of the Function	3-112
The FUNCTION Program Block	3-113
Data Type Declarations	3-113
Executable Statements	3-115
The END FUNCTION Keywords	3-116

The Prototype of the Function

The FUNCTION statement both declares and defines a 4GL function. The function declaration specifies the identifier of the function and the identifiers of its formal arguments (if any). These specifications are sometimes called the *function prototype*, as distinct from the *function definition*:



argument is the name of a formal argument to this function. This can be of any data type (page 3-67) except the ARRAY (page 3-71) data type.

function is the identifier that you declare for this 4GL function.

The Identifier of the Function

The *function* name must follow the rules for 4GL identifiers (page 2-9), and must be unique among all the names of functions or reports in the same program. If the name is also the name of a built-in 4GL function, an error occurs at link time, even if the program does not reference the built-in function. Like all 4GL identifiers, the name is not lettercase sensitive. For example, the function names **unIonized( )** and **Unionized( )** are identical to 4GL.

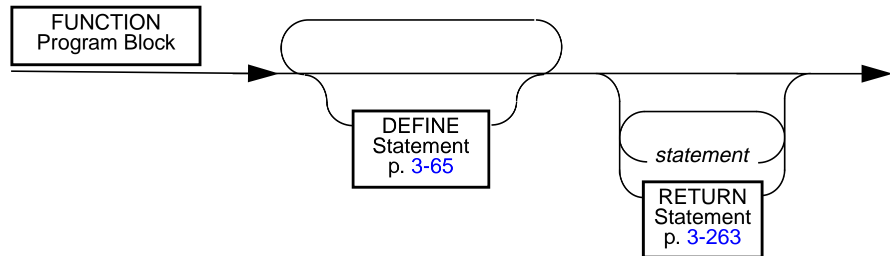
The Argument List of the Function

The names specified (between parentheses) in the argument list define the formal arguments, if any, as they will be received when the FUNCTION program block is executed. If no *argument* is specified, an empty argument list

must still be supplied. Argument names must be unique within the argument list of the current FUNCTION declaration. Their scope of reference is local to the function; that is, they are not visible in other program blocks.

The FUNCTION Program Block

The statements between the argument list and the END FUNCTION keywords comprise the FUNCTION program block. These statements are executed whenever the function is successfully invoked.



statement is any SQL statement or other 4GL statement (but not FUNCTION, REPORT, MAIN, GLOBALS, nor the report execution statements (NEED, PAUSE, PRINT, and SKIP).

Data Type Declarations

The data type of each formal argument of the function must be specified by a DEFINE statement that immediately follows the argument list. Any DEFINE declarations within a function definition must occur before any other statements within the FUNCTION program block. Just as in a MAIN or REPORT program block, a compile-time error occurs if any executable statement precedes a DEFINE declaration in the FUNCTION definition.

The actual argument in a call to the function need not be of the declared data type of the formal argument. If both are of compatible data types, **4GL** converts the actual argument to the data type that the function requires. If data type conversion is impossible, a run-time error occurs. For a discussion of compatible data types, see "Data Type Conversion" on page 3-319.

Here is an example of a call for which data conversion is necessary. The actual argument, the character string "105," must be converted to INTEGER.

```
DEFINE getStat INTEGER
LET getStat = getCustRec("105")
    . . .
FUNCTION getCustRec(cno)
    DEFINE cno, dno INTEGER
    . . .
    RETURN dno
END FUNCTION
```

The Function as a Local Scope of Reference

The same or a subsequent DEFINE statement must also declare any other local variable that is referenced in the same FUNCTION definition. Two local variables are declared in the previous example, the function argument, **cno**, and the variable named **dno**. The identifiers of local variables must be unique among the variables that are declared in the same FUNCTION definition. They are not visible in other program blocks.

Just as within MAIN or REPORT program blocks, statements in the function can reference previously declared module or global variables. Any module variable or global variable that has the same identifier as a local variable, however, is not visible within the scope of the local variable.

See the description of the DATABASE statement (page 3-59) for information about using the LIKE keyword during compilation to declare the data types of local variables indirectly. You can also use DATABASE within a FUNCTION definition to specify a new current database at run time (page 3-60).

Any GOTO or WHENEVER . . . GOTO statements in a function must reference a statement label (page 3-177) within the same FUNCTION program block.

Executable Statements

Any executable statements in the statement block are executed when the function is called. Here is a simple example of a function definition:

```
FUNCTION state_abbrev(state)
    DEFINE st LIKE state.code,
           state LIKE state.sname
    SELECT state.code INTO st FROM state
                WHERE state.sname MATCHES state
    RETURN st
END FUNCTION
```

In this example, the function definition contains two executable statements:

- DEFINE is a declarative statement that allocates storage in memory for the local variables **st** and **state**.
- SELECT is an executable SQL statement.
- RETURN returns control (and the value of **st**) to the calling routine.
- END FUNCTION (not executable) marks the end of the program block.

You can define a function whose statement block is empty. This enables the developer to test other parts of a program before a function definition is written, but such "dummy functions" are of limited use in a 4GL application.

Returning Values to the Calling Routine

Any programmer-defined 4GL function that returns one or more values to the calling routine must include the RETURN statement (page 3-263). Values specified in the RETURN statement must correspond in number and position, and must be of the same or of compatible data types (page 3-324), to the variables in the RETURNING clause of the CALL statement (page 3-19).

Unless it has the same name as a built-in operator (see Chapter 4), any built-in or programmer-defined function that returns a single value of a simple data type can appear in 4GL expressions (with its arguments, if any) if the returned value is of a range and data type that is valid in the expression:

```
DISPLAY AT 2,2 ERR_GET(SQLCA.SQLCODE)
```

Returned Character Strings

4GL allocates 5 kilobytes of memory to store character strings returned by functions, in 10 blocks of 512 bytes. A returned CHAR value can be no larger than 511 bytes (because every string requires a terminating ASCII 0), and no more than 10 of these 511-byte strings can be returned. (This restriction has no effect on TEXT arguments, which are passed by reference, not by value.)

Since no value can occupy more than one block, ten returned strings of 256 bytes would leave no room for an eleventh. Similarly, if there are partially evaluated string expressions in the calling sequence, then some space for returned values may already be in use, as in this example:

```
FUNCTION func_f( )
   LET g = func_h( ), func_i( )
END FUNCTION
```

While 4GL evaluates either **func_h( )** or **func_i( )**, the returned value from the other occupies part of the temporary string space. If insufficient space is available in memory for a returned string, 4GL issues run-time error -4518.

The END FUNCTION Keywords

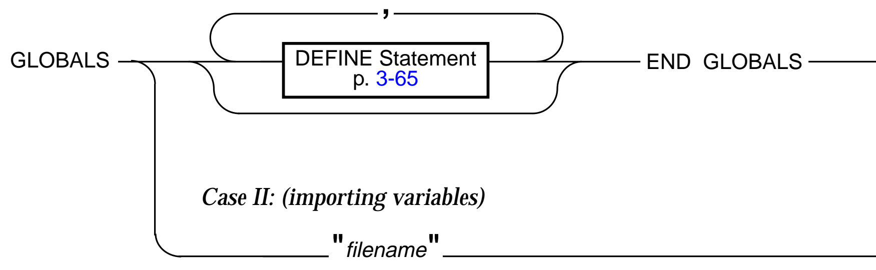
The END FUNCTION keywords mark the end of the FUNCTION program block. In this release of INFORMIX-4GL, only another FUNCTION definition or else the REPORT statement can follow the END FUNCTION keywords.

References

CALL, DEFINE, RETURN, WHENEVER

GLOBALS

The GLOBALS statement declares modular variables that can be exported to other program modules. It can also import variables from other modules.



filename is a quoted string that specifies the pathname of a file that contains the GLOBALS . . . END GLOBALS statement (and optionally the DATABASE statement, but no executable statement). The *filename* can include a pathname. The **.4gl** file extension is required.

Usage

In general, a program variable is visible only in the same FUNCTION, MAIN, or REPORT program block in which it was declared. To make its scope of reference the entire source module, you must specify a *modular* declaration, by locating the DEFINE statement outside of any program block.

To extend the visibility of one or more modular variables beyond the source module in which they are declared, you must take the following steps:

- Declare the variable in a GLOBALS . . . END GLOBALS declaration in a **.4gl** file that contains only GLOBALS statements.
- Specify that file in GLOBALS "*filename*" statements in each additional source module that includes statements referencing the variable.

These files must also be compiled and linked with the 4GL application.

Declaring and Exporting Global Variables

To declare global variables, the GLOBALS statement must appear before the first MAIN, FUNCTION, or REPORT program block, so that variables that you declare in the GLOBALS statement are modular in their scope of reference. You can include one or more DEFINE statements after the GLOBALS keyword. The END GLOBALS keywords must follow the last DEFINE declaration.

If you use the LIKE keyword in the DEFINE declaration, then the DATABASE statement must precede the GLOBALS statement within the same module.

The following program fragment declares a *global record*, a *global array*, and a simple *global variable* that are referenced by built-in and programmer-defined functions in the same source code module:

```
DATABASE stores2
GLOBALS
    DEFINE p_customer RECORD LIKE customer.*,
            p_state ARRAY[fifty] OF RECORD LIKE state.*,
            fifty, state_cnt  SMALLINT
END GLOBALS
MAIN
    ...
END MAIN
FUNCTION get_states()
    ...
    FOREACH c_state INTO p_state[state_cnt].*
        LET state_cnt = state_cnt + 1
        IF state_cnt > fifty THEN
            EXIT FOREACH
        END IF
    END FOREACH
    ...
END FUNCTION
FUNCTION statehelp()
    DEFINE idx SMALLINT
    ...
    CALL SET _COUNT(state_cnt)
    DISPLAY ARRAY p_state TO s_state.*
    LET idx = ARR_CURR()
    CLOSE WINDOW w_state
    LET p_customer.state = p_state[idx].code
    DISPLAY BY NAME p_customer.state
    RETURN
END FUNCTION
```

A compile-time error would occur if you declared a 4GL variable of *modular* scope called **fifty**, **p_customer**, **p_state**, or **state_cnt** in the same module as this GLOBALS statement. (If you want, however, you can declare *local* variables whose names match those of variables from GLOBALS declarations.)

Importing Global Variables

A *globals file* is a source module that contains a GLOBALS . . . END GLOBALS statement. This can also contain a DATABASE statement (page 3-59), but no executable statements. The scope of reference of variables declared in that file can be extended to all the program blocks of any 4GL program module that includes a GLOBALS "*filename*" statement, for *filename* the globals file.

To import global variables into other modules, you must do the following:

1. Create a globals file called *filename*.**4gl** that includes the following:.
   - If necessary, include a DATABASE statement. (This is required only if you use the LIKE keyword in the DEFINE declaration. If present, the DATABASE statement must precede the GLOBALS statement. For the syntax of *column* qualifiers, see "Indirect Typing" on page 3-69.)
   - Then include the GLOBALS keyword, followed by as many DEFINE statements as necessary to declare your global variables. (You can also include no DEFINE statements, if the GLOBALS "*filename*" statement is used only to apply a DATABASE statement to several modules.)
   - Finally, include the END GLOBALS keywords.
2. In any other module of the program that includes statements referencing the global variables, include a GLOBALS "*filename*" statement before the first MAIN, FUNCTION, or REPORT program block. Specify the "*filename*" of the globals file, but do not include the END GLOBALS keywords.

These two steps correspond, respectively, to Case I and Case II in the syntax diagram on page 3-117. For example, the globals file **d4_glob.4gl** in the stores demonstration application includes the following DATABASE and GLOBALS statements:

```
DATABASE stores
GLOBALS
    DEFINE
        p_customer RECORD LIKE customer.*,
        p_orders RECORD
            order_num LIKE orders.order_num,
            order_date LIKE orders.order_date,
            po_num LIKE orders.po_num,
            ship_instruct LIKE orders.ship_instruct
        END RECORD,
        p_items ARRAY[10] OF RECORD
            item_num LIKE items.item_num,
            stock_num LIKE items.stock_num,
            manu_code LIKE items.manu_code,
            description LIKE stock.description,
            quantity LIKE items.quantity,
            unit_price LIKE stock.unit_price,
            total_price LIKE items.total_price
        END RECORD,
        p_stock ARRAY[30] OF RECORD
            stock_num LIKE stock.stock_num,
            manu_code LIKE manufact.manu_code,
            manu_name LIKE manufact.manu_name,
            description LIKE stock.description,
            unit_price LIKE stock.unit_price,
            unit_descr LIKE stock.unit_descr
        END RECORD,
        p_state ARRAY[fifty] OF RECORD LIKE state.*,
        fifty, state_cnt, stock_cnt INTEGER,
        print_option CHAR(1)
END GLOBALS
```

The next program fragment include a GLOBALS statement that specifies **d4_glob.4gl** as the globals file that declares global variables:

```
GLOBALS "d4_glob.4gl"
MAIN
    DEFER INTERRUPT
    ...
    CALL get_states()
    CALL get_stocks()
    ...
END MAIN
```

Here the database specified by the DATABASE statement in the globals file is both the default database at compile time (page 3-59) and the current database at run time (page 3-60), because the GLOBALS "**d4_glob.4gl**" statement includes the DATABASE statement before the MAIN program block.

If a *local* variable has the same name as another variable that you declare in the GLOBALS statement, only the local variable is visible within its scope of reference. Similarly, a *modular* variable takes precedence in the module where it is declared over any variable of the same name whose declaration is in the *filename* referenced by a GLOBALS statement. (A compile-time error occurs if you declare another module variable with the same identifier as another variable that the GLOBALS ... END GLOBALS statement declares in the same module.) For more information about the scope and visibility of 4GL identifiers, see "4GL Identifiers" on page 2-9.

References

DATABASE, DEFINE, FUNCTION, INCLUDE, MAIN, REPORT

GOTO

The GOTO statement transfers program control to a labelled line within the same program block.

```
GOTO ──┬─────┬── label name ──┤
       └─ : ─┘
```

label name is a *statement label* that you declare in a LABEL statement.

Usage

The GOTO statement transfers control of execution within a program block. Upon encountering this statement, 4GL jumps to the statement immediately following the specified LABEL statement (page 3-177), and resumes execution there, skipping any intervening statements that lexically follow the GOTO statement. These rules apply to the use of the GOTO and LABEL statements:

- To transfer control to a labelled line, the GOTO statement must use the same *label name* as the LABEL statement above the desired line.
- Both statements must reside in the same MAIN, FUNCTION, or REPORT block. You cannot use GOTO to transfer into or out of a program block.

Excessive use of GOTO statements in 4GL (or any programming language) can make your code difficult to read or to maintain, or can result in a loop that has no termination. Many situations in which you need to transfer control of program execution can be solved by using one of the following alternatives to the GOTO statement:

- Boolean expressions and the CASE, FOR, IF, and WHILE statements.
- The EXIT keyword in blocks within the following statements:

BEGIN	FOR	INPUT ARRAY
CASE	FOREACH	MENU
DISPLAY ARRAY	INPUT	WHILE

- The CONTINUE keyword in blocks within the following statements:

FOR	INPUT	MENU
FOREACH	INPUT ARRAY	WHILE

- The CALL statement.
- The WHENEVER statement.

It is convenient to use the GOTO and LABEL statements in some situations; for example, to exit from deeply nested code:

```
FOR i = 1 TO 10
    FOR j = 1 TO 20
        FOR k = 1 To 30
            ...
            IF pa_array3d[i,j,k] IS NULL THEN
                GOTO :done
            ELSE
                ...
            END IF
            ...
        END FOR
    END FOR
END FOR

LABEL done:
ERROR "Cannot complete processing."
ROLLBACK WORK
```

More important than avoiding the GOTO statement, however, is to adhere to the design principle that any block of statements (such as a function or a loop) have only *one entry point* and *one exit point*, as in this program fragment:

```
CALL do_things(value)          --invokes a FUNCTION block
...
FUNCTION do_things(arglist)    --unique entry point
        ...
    IF (exit_condition) THEN
        GOTO :outofhere        --jump within same program block
    END IF
    ...
    LABEL outofhere:
    CALL clean_up()
    RETURN ret_code            --unique exit point
END FUNCTION                   --marks end of FUNCTION construct
```

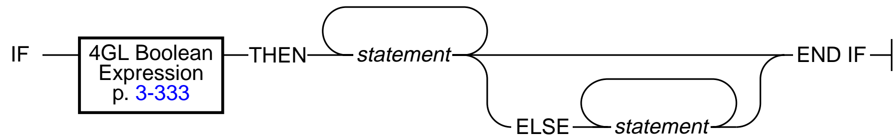
You can optionally place a colon (:) before *label name* in the GOTO statement. This conforms to the ANSI standard for embedded SQL syntax, but has no impact on the execution of the GOTO statement.

References

CASE, FOR, IF, FOR, FUNCTION, LABEL, MAIN, REPORT, WHENEVER, WHILE

IF

The IF statement executes a group of statements conditionally. It can switch program control conditionally between two blocks of statements.



statement is an SQL statement or other 4GL statement.

Usage

If the Boolean expression is TRUE, 4GL executes the block of statements following the THEN keyword, until it reaches either the ELSE keyword or the END IF keywords. 4GL then resumes execution after the END IF keywords.

If the Boolean expression is FALSE, 4GL executes the block of statements between the ELSE keyword and the END IF statement. If you omit the ELSE keyword, 4GL resumes execution with the statement following the END IF keywords. The Boolean expression is treated as FALSE if it contains a NULL value anywhere (except as the operand of the IS NULL operator).

If you have a set of nested IF statements, all testing the same value, consider using a CASE statement. In the following example, if **direction** matches BACK, then 4GL decrements **p_index** by one. If **direction** matches the string **FORWARD**, then 4GL increments **p_index** by one.

```
IF  direction = "BACK" THEN
    LET p_index = p_index - 1
    DISPLAY dp_stock[p_index].* TO s_stock.*
    ELSE IF direction = "FORWARD" THEN
            LET p_index = p_index + 1
            DISPLAY dp_stock[p_index].* TO s_stock.*
        END IF
END IF
```

NLS

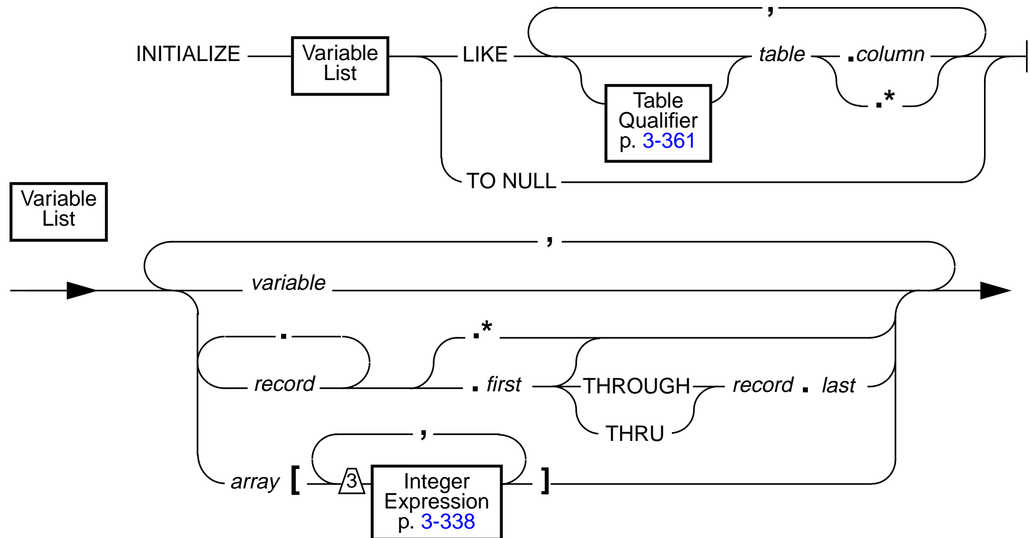
When NLS is active, the evaluation of less than (<) and greater than (>) expressions containing character arguments is dependent on the LC_COLLATE setting in the user environment.

References

CASE, FOR, WHENEVER, WHILE

INITIALIZE

The INITIALIZE statement assigns initial NULL or default values to variables.



array	is the name of a variable of the ARRAY data type.
column	is the name of a column of *table* for which a DEFAULT value exists.
first	is the name of a member variable to be initialized.
last	is another member of *record* that was declared later than *first.*
record	is the name of a variable of the RECORD data type.
table	is the name or synonym of the table or view that contains *column.*
variable	is the name of a variable of a simple data type (page 3-68).

Usage

After you declare a variable with a DEFINE statement, the Source Compiler allocates memory to that variable. The contents of the variable, however, is whatever happens to occupy that memory location. The INITIALIZE statement can specify initial values for 4GL variables in either of two ways:

- The LIKE keyword assigns the *default* values of a specified database *column*, using default values from the **syscolval** table (page 3-126).
- You can use the TO NULL keywords to assign NULL values, using the representation of NULL for the declared data type of each *variable.*

The LIKE Clause

The LIKE clause specifies default values from one or more **syscolval** columns:

Just as in the DEFINE or VALIDATE statement, the LIKE clause requires a DATABASE statement to specify a default database (page 3-59) at compile time. The DATABASE statement to specify a default database must precede the first program block in the same module as the INITIALIZE statement.

When initializing variables with the default values of database columns, the variables must match the columns in order, number, and data type. You must prefix the name of each column with the name of its table. For example, the following statement assigns to three variables the default values from three database columns in table **tab1**:

```
INITIALIZE  var1,  var2,  var3
    LIKE  tab1.col1,  tab1.col2,  tab1.col3
```

The *table*`.*` notation specifies every column in the specified table. If **tab1** has only the three columns (**col1**, **col2**, and **col3**), then the following statement is equivalent to the previous one:

```
INITIALIZE  v_cust.*  LIKE  customer.*
```

ANSI

In an ANSI-compliant database, you must qualify each *table* name with that of its owner (*owner. table*), if the application will be run by a user who does not own the table. For example, if you own **tab1**, and Lydia owns **tab2**, and Boris owns **tab3**, then the following statement is valid:

```
INITIALIZE  var1,  var2,  var3
    LIKE  tab1.var1,  lydia.tab2.var2,  boris.tab3.var3
```

You can include the *owner* name as a prefix in a database that is not ANSI-compliant, but if the owner name that you specify is incorrect, you receive an error. For additional information, see the *Informix Guide to SQL: Reference.*

The INITIALIZE statement looks up the default values for database columns in the DEFAULT column of the **syscolval** table in the default database. Any changes to **syscolval** after compilation have no effect on the 4GL program, unless you recompile the program.

To enter default values in this table, use the **upscol** utility, as described in Appendix B. If a column has no default value in the **syscolval** table, 4GL assigns NULL values to any variables initialized from that column. If the database is not ANSI-compliant, **upscol** creates a single **syscolval** table.

ANSI

In an ANSI-compliant database, each user can create an *owner*.**syscolval** table, which sets the default values only for the tables owned by that user. If you omit the owner of the table and you own the table, your **syscolval** table becomes the source for the defaults when you compile the program. If the *owner*.**syscolval** table does not exist, the LIKE clause of the INITIALIZE statement sets the values of the specified variables to NULL

OL

You cannot use **upscol** to specify attributes nor validation criteria for TEXT or BYTE columns. Therefore, you cannot use the LIKE clause of the INITIALIZE statement to assign non-NULL values to variables of these large binary data types.

The TO NULL Clause

Use the TO NULL clause to assign a NULL value to a variable. The following statement initializes all variables in the **v_orders** record to NULL:

```
INITIALIZE  v_orders.*  TO  NULL
```

You may wish to initialize variables to NULL for the following reasons:

- To assign an initial value to a variable, rather than leaving it unassigned.
- To discard some existing value of a variable. This may be convenient if you want to reuse the same variable later in a program.

To optimize performance, you may wish to limit the use of this statement. For example, the next program fragment uses INITIALIZE once to create a NULL record, and then uses the LET statement to initialize another record:

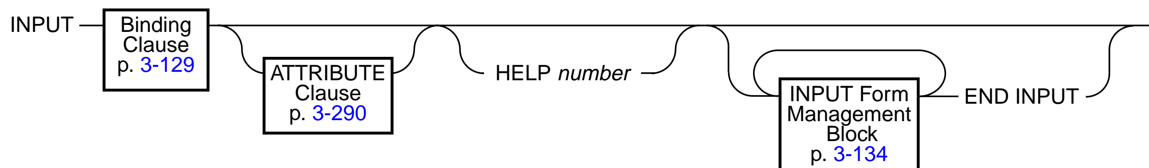
```
DATABASE stores2
MAIN
DEFINE p_customer, n_customer RECORD LIKE customer.*
    INITIALIZE n_customer.* TO NULL
    LET p_customer.* = n_customer.*
```

References

DATABASE, DEFINE, GLOBALS, LET, VALIDATE

INPUT

The INPUT statement supports data entry by users into fields of a screen form.



number is a literal integer (page 3-340) to specify a Help message number.

Usage

The INPUT statement assigns to one or more variables the values users enter into the fields of a screen form. This statement can include *statement blocks* to be executed under conditions that you specify, such as screen cursor movement, or other user actions. To use this statement, you must do the following:

1. Specify screen field(s) in a form specification file, and compile the form.
2. Declare variable(s) with the DEFINE statement.
3. Open and display the screen form in either of the following ways:
   - The OPEN FORM and DISPLAY FORM statements.
   - An OPEN WINDOW statement that uses the WITH FORM keywords.
4. Use the INPUT statement to assign values to the variables from data that the user enters into fields of the screen form.

When the INPUT statement is encountered, 4GL does the following:

1. Displays any default values in the screen fields, unless you specify the WITHOUT DEFAULTS keywords (as described on page 3-131.)
2. Moves the cursor to the first field explicitly or implicitly referenced in the binding clause, and waits for the user to enter a value in that field.
3. Assigns the user-entered field value to a corresponding program variable when the user moves the cursor from the field or presses the Accept key.

The INPUT statement activates the *current form* (the form that was most recently displayed, or the form in the current 4GL window). When the INPUT statement completes execution, the form is deactivated. After the user presses the Accept key, the INSERT statement of SQL can insert values from the program variables into the appropriate database tables.

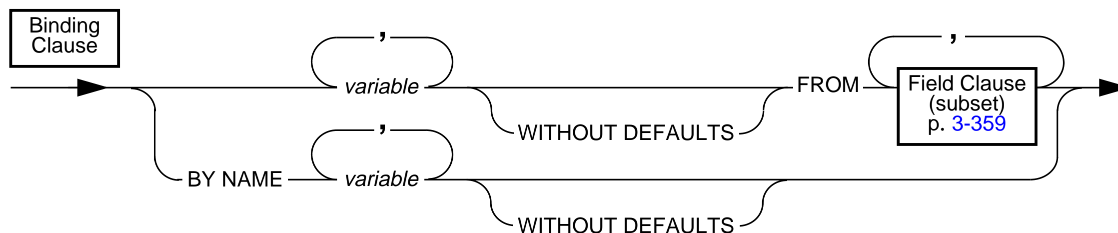
The following topics are described in this section:

Topic	Page
The Binding Clause	3-129
The ATTRIBUTE Clause	3-133
The HELP Clause	3-133
The INPUT Form Management Blocks	3-134
The EXIT INPUT Statement	3-143
The END INPUT Keywords	3-144
Using Built-In Functions and Operators	3-144
Keyboard Interaction	3-146
Editing Keys	3-147
Using Large Data Types	3-149
Completing the INPUT Statement	3-150

The Binding Clause

The *binding clause* temporarily associates fields of the screen form with 4GL variables, so that the 4GL program can manipulate values that the user enters in the form. INPUT statements supports two types of binding clauses:

- In the special case where all of the variables have names that are identical (apart from qualifiers) to the names of fields, you can specify INPUT BY NAME *variable list* to bind the specified variables to their namesake fields implicitly. (See also page 3-131.)
- In the general case, you can specify INPUT *variable list* FROM *field list* to bind variables explicitly to fields.



variable is the name of a variable to store values entered in the field.

Here *variable* supports the syntax of a *receiving variable* (page 3-178) in the LET statement, but you can also use *record* . * or the THRU or THROUGH notation to specify all or some of the members of a program record.

The field names must be among those declared in the ATTRIBUTES section of the form specification file of the current form. These can include simple fields, members of screen records, multiple-segment WORDWRAP fields, and FORMONLY fields, but cannot include records from screen arrays.

The Correspondence of Variables and Fields

The total number of variables in the variable list must equal the total number of fields that the FROM clause specifies (or that the BY NAME clause implies).

The order in which the screen cursor moves from field to field in the form is determined by the order of the field names in the FROM clause, or else by the order of variable names in the BY NAME clause. (See also the NEXT FIELD keywords on page 3-141, and the WRAP and FIELD ORDER options of the OPTIONS statement on page 3-232.)

Each screen field and its corresponding variable must have the same (or a compatible) data type. When the user enters data in a field, 4GL checks the value against the data type of the *variable*, not that of the field. You must first declare all the variables before using the INPUT statement.

The binding clause can specify variables of any 4GL data type. If a variable is declared LIKE a SERIAL column, however, then 4GL does not allow the screen cursor to stop in the field. (Values in SERIAL columns are maintained by the database engine, not by 4GL.)

Displaying Default Values

If you omit the WITHOUT DEFAULTS keywords, 4GL displays default values from the program array when the form is activated. 4GL determines the default values in the following way, in descending order of precedence:

1. The DEFAULT attribute (from the form specification file).
2. The DEFAULT column value (from the **syscolval** table).

4GL assigns NULL values to all variables for which no default is set. But if you include the WITHOUT NULL INPUT option in the DATABASE section of the form specification file, then 4GL assigns the following default values:

Field Type	Default
character	blank (= ASCII 32)
number	0
INTERVAL	0
MONEY	$0.00
DATE	12/31/1899
DATETIME	1899-12-31 23:59:59.99999

The WITHOUT DEFAULTS Keywords

If you specify the WITHOUT DEFAULTS option, however, the screen displays the *current* values of the variables when the INPUT statement begins. This option is available with both the BY NAME and the FROM binding clauses.

To display initialized values, rather than defaults, you can do the following:

1. Initialize the variables with whatever values you want to display.
2. Use INPUT ... WITHOUT DEFAULTS to display the current values of the variables, and to allow the user to change those values.

The following INPUT statement causes 4GL to display the character string "`Send via air express`" in the **ship_instruct** field:

```
LET pr_orders.ship_instruct = "Send via air express"
INPUT BY NAME pr_orders.order_date THRU pr_orders.paid_date
    WITHOUT DEFAULTS
END INPUT
```

The WITHOUT DEFAULTS option is useful when you want the user to be able to make changes to existing rows of the database. You can display the existing database values on the screen before the user begins editing the data. The FIELD_TOUCHED( ) operator (described briefly on page 3-144, and in detail on page 4-64) can help you to determine which fields have been altered, and that therefore require updates to the database.

The BY NAME Clause

The BY NAME clause implicitly binds the fields to the 4GL variables that have the same identifiers as field names. You must first declare variables with the same names as the fields from which they accept input. 4GL ignores any *record name* prefix when making the match.

The unqualified names of the variables and of the fields must be unique and unambiguous within their respective domains. If they are not, 4GL generates a runtime error, and sets the **status** variable to a negative value. (To avoid this error, use the FROM clause instead of the BY NAME clause when the screen fields and the variables have different names.)

The user can enter values only into fields that are implied in the BY NAME clause. For example, the INPUT statement in the following example specifies variables for all the screen fields except **customer_num**:

```
DEFINE pr_customer RECORD LIKE customer.*
...
INPUT BY NAME pr_customer.fname, pr_customer.lname,
    pr_customer.company, pr_customer.address1,
    pr_customer.address2, pr_customer.city, pr_customer.state,
    pr_customer.zipcode, pr_customer.phone
```

Because **pr_customer.customer_num** does not appear in the list of variables, the user cannot enter a value for it. A functionally equivalent statement is:

```
DEFINE pr_cust RECORD LIKE customer.*
...
INPUT BY NAME pr_cust.fname THRU pr_cust.phone
```

The FROM Clause

When variables and fields do not have the same names, you must use the FROM clause. For any INPUT statement with FROM, you must specify the same number of variables and fields, and list them in the same order on both sides of the FROM keyword. The user can position the cursor only in fields that you include explicitly or implicitly in the FROM clause. These fields must correspond both in order and in number to the list of variables, and must be of the same or compatible data types as the corresponding variables:

```
DEFINE pr_cust RECORD LIKE customer.*
...
INPUT pr_cust.fname, pr_cust.lname FROM fname, lname
```

The THRU (or THROUGH) keyword implicitly includes the variables between two specified member variables of a program record. For example, the next statement maps fields to all member variables from **fname** to **phone**:

```
INPUT pr_cust.fname THRU pr_cust.phone
    FROM fname, lname, company, address1,
        address2, city, state, zipcode, phone
```

If the form specification file declared a screen record as **fname** THRU **phone**, then you can abbreviate this statement even further:

```
INPUT pr_cust.fname THRU pr_cust.phone FROM sc_cust.*
```

Note: *You cannot use the* THRU *nor* THROUGH *keywords in the* FROM *clause.*

The ATTRIBUTE Clause

For the syntax of the ATTRIBUTE clause, see page 3-290. This section describes the use of the ATTRIBUTE clause within an INPUT statement.

If you specify form attributes with the INPUT statement, the new attributes apply only during the current activation of the form. When actions of the user deactivate the form, the form reverts to its previous attributes. The following INPUT statement assigns the RED and REVERSE attributes:

```
INPUT  p_addr.* FROM sc_addr.* ATTRIBUTE (RED, REVERSE)
```

This statement assigns the WHITE attribute:

```
INPUT BY NAME p_items ATTRIBUTE (WHITE)
```

The ATTRIBUTE clause overrides any default display attributes specified in an OPTIONS or OPEN WINDOW statement for these fields. It also suppresses any default attributes specified in the **syscolatt** table of the **upscol** utility.

The HELP Clause

The HELP clause includes a literal integer (page 3-340) to specify the *number* of the Help message to display. The Help message is displayed in the Help window, as described on page 2-22. This window appears if the user presses the Help key while the screen cursor is in any field that you listed in the FROM clause, or that you implied in the BY NAME clause. The default Help key is CONTROL-W, but you can specify a different Help key by using the OPTIONS statement (page 3-233).

This example specifies Help message 311 if the user requests Help from any field in the **s_items** screen array:

```
INPUT p_items.* FROM s_items.* HELP 311
```

The next example tells 4GL to display message 12 if the user presses the Help key when the screen cursor is in either of two fields:

```
INPUT cust.fname, cust.lname FROM fname, lname HELP 12
```

You create Help messages in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement. Use the **mkmessage** utility, as described in Appendix B, to create a run-time version of the Help file. A run-time error occurs in the following situations:

- 4GL cannot open the Help file.
- You specify a *number* that is not in the Help file.
- You specify a *number* outside the range from -32,767 to 32,767.

The Help message corresponding to your HELP clause specification applies to the entire INPUT statement. To override this with field-level Help messages, specify the Help key in an ON KEY block (page 3-137) that invokes the INFIELD( ) operator and SHOWHELP( ) function.

If you provide messages to assist the user through an ON KEY clause, rather than by the HELP clause, the messages must be displayed in a 4GL window within the 4GL screen, rather than in the separate Help window.

The INPUT Form Management Blocks

Each INPUT form management block includes a *statement block* of at least one statement, and an *activation clause* that specifies when to execute the *statement block*. An INPUT form management block can specify any of the following:

- The statements to execute before or after visiting specific screen fields.
- The statements to execute when the user presses a key sequence.
- The statements to execute before or after the INPUT statement.
- The next field to which to move the screen cursor.
- When to terminate execution of the INPUT statement.

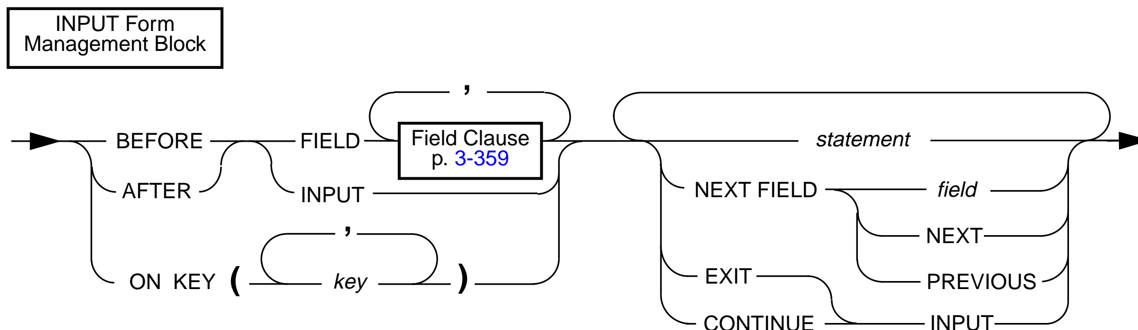
The *activation clause* can specify any one of the following:

- Pre- and post-INPUT actions (the BEFORE or AFTER INPUT clause).
- Keyboard sequence conditions (the ON KEY clause).
- Cursor movement conditions (the BEFORE or AFTER FIELD clause).

The *statement block* can include any SQL or 4GL statements, as well as:

- Cursor movement instructions (the NEXT FIELD clause).
- Termination of the INPUT statement (the EXIT INPUT statement).
- Returning control to the user, without terminating the INPUT statement (the CONTINUE INPUT statement).

The *activation clause* and the *statement block* correspond respectively to the left-hand and right-hand elements in the following syntax diagram:



field in the name of a field (page 3-359) in the current form.

key is one or more keywords to specify physical or logical keys. For details, see "The ON KEY Block" on page 3-137.

statement is an SQL statement or other 4GL statement.

After BEFORE FIELD, AFTER FIELD, or NEXT FIELD, the Field clause specifies a field that the binding clause referenced implicitly (in the BY NAME clause, or as *record*.* or *array* [*line*].*) or explicitly. You can qualify a field name by a *table* reference, or the name of a *screen record* or a *screen array* or *array* [*line*].

If you include one or more form management blocks, the END INPUT keywords must terminate the INPUT statement. If no form management block is included, 4GL waits while the user enters values into the fields. When the user accepts the values in the form, the INPUT statement terminates.

If you include a form management block, 4GL executes or ignores the statements in a form management block, depending on the following:

- Whether you specify the BEFORE INPUT or AFTER INPUT keywords.
- The *fields* to which and from which the user moves the screen cursor.
- The *keys* that the user presses.

4GL deactivates the form while executing statements in a form management block. After executing the statements, 4GL reactivates the form, allowing the user to continue entering or modifying the data values in fields.

The Precedence of INPUT Form Management Blocks

This is the order in which 4GL executes the statements from control blocks:

1. BEFORE INPUT
2. BEFORE FIELD
3. ON KEY
4. AFTER FIELD
5. AFTER INPUT

You can list these blocks in any order. If you develop some consistent ordering, however, your code may be easier to read.

Within these blocks, you can include the NEXT FIELD keywords and EXIT INPUT statement, as well as most 4GL and SQL statements.You cannot specify a CONSTRUCT, PROMPT, INPUT, or INPUT ARRAY statement, but you can invoke a function that executes one or more of these statements.

The activation clauses that you can specify in form management blocks are described in their order of execution by 4GL. Descriptions of NEXT FIELD and EXIT INPUT follow the discussions of these activation clauses. No subsequent INPUT control block statements are executed if EXIT INPUT executes.

The BEFORE INPUT Block

You can use the BEFORE INPUT block to display messages on how to use the INPUT statement. For example, the following INPUT statement fragment displays a message informing the user how to enter data into the table:

```
INPUT BY NAME p_customer.*
    BEFORE INPUT
        DISPLAY "Press ESC to enter data" AT 1,1
```

4GL executes the BEFORE INPUT block after displaying the default values in the fields and before letting the user enter any values. (If you included the WITHOUT DEFAULTS clause, 4GL displays the current values of the variables, not the default values, before executing the BEFORE INPUT block.)

An INPUT statement can include only one BEFORE INPUT block. You cannot include the FIELD_TOUCHED( ) operator in the BEFORE INPUT block.

The BEFORE FIELD Block

4GL executes the statements in the BEFORE FIELD block associated with a field whenever the cursor moves into the field, but before the user enters a value. You can specify no more than one BEFORE FIELD block for each field.

The following program fragment defines two BEFORE FIELD blocks. When the cursor enters the **fname** or **lname** field, 4GL displays a message:

```
BEFORE FIELD fname
    MESSAGE  "Enter first name of customer"
BEFORE FIELD lname
    MESSAGE  "Enter last name of customer"
```

You can use a NEXT FIELD clause within a BEFORE FIELD block to restrict access to a field. You can also use a DISPLAY statement within a BEFORE FIELD block to display a default value in a field.

The following statement fragment causes 4GL to prompt the user for input when the cursor is in the **stock_num**, **manu_code**, or **quantity** fields:

```
INPUT p_items.* FROM s_items.*
    BEFORE FIELD stock_num
        MESSAGE "Enter a stock number."
    BEFORE FIELD manu_code
        MESSAGE "Enter the code for a manufacturer."
    BEFORE FIELD quantity
        MESSAGE "Enter a quantity."
    ...
END INPUT
```

The ON KEY Block

Statements in the ON KEY block are executed if the user presses some key that you specify by these keywords (in lowercase or uppercase letters):

ACCEPT	HELP	NEXT *or*	RETURN
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS *or*	TAB
ESC *or* ESCAPE	LEFT	PREVPAGE	UP

F1 *through* F64

CONTROL-*char* (except A, D, H, I, J, L, M, R, or X)

For example, the following ON KEY block displays a Help message. The BEFORE INPUT clause informs the user how to access Help:

```
BEFORE INPUT
    DISPLAY "Press CONTROL-W or CTRL-F for Help"
ON KEY (CONTROL-W, CONTROL-F)
    CALL customer_help()
```

The next statement defines an ON KEY block for the CONTROL-B key. Whenever the user presses the CONTROL-B key, 4GL determines if the screen cursor is in the **stock_num** or **manu_code** fields. If it is in either one of these fields, then 4GL calls the **stock_help( )** function and sets **quantity** as the next field.

```
INPUT p_items.* FROM s_items.*
    ON KEY (CONTROL-B)
        IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            CALL stock_help()
            NEXT FIELD quantity
        END IF
```

Some keys require special consideration if specified in an ON KEY block:

Key	**Special Considerations**
ESC or ESCAPE	You must specify another key as the Accept key in the OPTIONS statement, because this is the default Accept key.
INTERRUPT	You must execute a DEFER INTERRUPT statement. If the user presses the Interrupt key under these conditions, **4GL** executes the statements in the ON KEY block and sets **int_flag** to nonzero, but does not terminate the INPUT statement. **4GL** also executes the statements in this ON KEY block if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, **4GL** sets **quit_flag** to non-zero.
CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these keys for field editing.
I, J, and M	The regular meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is lost to the user. Instead, the key is trapped by 4GL and used to activate the ON KEY block. For example, if CONTROL-M appears in an ON KEY block, the user cannot press RETURN to advance the cursor to the next field. If you include one of these keys in an ON KEY block, be careful to restrict the scope of the block to specific fields.

You may not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

If you use the OPTIONS statement to redefine the Accept or Help keys, the keys assigned to these sequences cannot be used in an ON KEY clause. For example, if you redefine the Accept key by using the following statement, you should not define an ON KEY block for the key sequence CONTROL-B:

```
OPTIONS ACCEPT KEY (CONTROL-B)
```

When the user presses CONTROL-B, 4GL will always perform the Accept key function, regardless of the presence of an ON KEY (CONTROL-B) block.

If the user activates an ON KEY block while entering data in a field, 4GL:

1. Suspends input to the current field.
2. Preserves the input buffer that contains the characters the user has typed.
3. Executes the statements in the current ON KEY block.
4. Restores the input buffer for the current screen field.
5. Resumes input in the same field, with the screen cursor at the end of the buffered list of characters.

You can change this default behavior by performing the following tasks in the ON KEY block:

- Resuming input in another field by using the NEXT FIELD statement.
- Changing the input buffer value for the current field by assigning a new value to the corresponding variable, and then displaying this value.

This block can support *accelerator keys* for common functions, such as saving and deleting. You can use the INFIELD( ) operator in the ON KEY clause to support field-specific actions. For example, you can implement field-level Help by using the INFIELD( ) operator and the built-in SHOWHELP( ) function.

The AFTER FIELD Block

4GL executes the statements in the AFTER FIELD block associated with a field every time the cursor leaves the specified field. Any of the following keys can cause the cursor to leave the field.

- Home or End key
- Any arrow key
- RETURN or TAB key
- Accept key
- Interrupt or Quit key (if a supporting DEFER statement was included)

You can specify only one AFTER FIELD block for each field.

This AFTER FIELD block checks if the **stock_num** and **manu_code** fields contain values. If they contain values, 4GL calls the **get_item( )** function:

```
AFTER FIELD stock_num, manu_code
    LET pa_curr = ARR_CURR()
    IF p_items[pa_curr].stock_num IS NOT NULL
        AND p_items[pa_curr].manu_code IS NOT NULL THEN
            CALL get_item()
            IF p_items[pa_curr].quantity IS NOT NULL THEN
                CALL get_total()
            END IF
    END IF
```

The following INPUT statement performs a NULL test to determine whether the user enters a value in the **address1** field, and returns to that field if no value was entered:

```
INPUT p_addr.* FROM sc_addr.*
    AFTER FIELD address1
        IF p_addr.address1 IS NULL THEN
            NEXT FIELD address1
        END IF
END INPUT
```

The user terminates the INPUT statement by pressing the Accept key when the cursor is in any field, or by pressing the TAB or RETURN key after the *last* field. You can use the AFTER FIELD block on the last field to override this default termination. (Including the INPUT WRAP in the OPTIONS statement produces the same effect.)

When the NEXT FIELD keywords appear in an AFTER FIELD block, the cursor moves to in the specified field. If an AFTER FIELD block appears for each field, and NEXT FIELD keywords are in each block, the user cannot leave the form.

The AFTER INPUT Block

4GL executes the AFTER INPUT block when the user presses the Accept key. You can use the AFTER INPUT block to validate, save, or alter the values the user entered by using the built-in GET_FLDBUF( ) or FIELD_TOUCHED( ) operators within the AFTER INPUT clause. (Use of these operators in an INPUT

statement is described in"Using Built-In Functions and Operators" on page 3-144.) The next example uses the AFTER INPUT block to require that a first name be specified for any customers with the last name `Smith`:

```
INPUT BY NAME p_customer.fname THRU p_customer.phone
    AFTER INPUT
        IF p_customer.lname="Smith" THEN
            IF NOT FIELD_TOUCHED(p_customer.fname) THEN
                CALL mess("You must enter a first name.")
                NEXT FIELD fname
            END IF
        END IF
    END INPUT
```

4GL executes the AFTER INPUT block only when the INPUT statement is terminated by the user choosing one of the following keys:

- The Accept key.
- The Interrupt key (if the DEFER INTERRUPT statement has executed).
- The Quit key (if the DEFER QUIT statement has executed).

The AFTER INPUT clause is not executed in the following situations:

- The user presses the Interrupt or Quit key when the DEFER INTERRUPT or DEFER QUIT statement, respectively, has not executed. In either case, the program terminates immediately.
- The EXIT INPUT statement terminates the INPUT statement.

You can place the NEXT FIELD clause in this block to return the cursor to the form. If you place a NEXT FIELD clause in the AFTER INPUT block, use it in a conditional statement. Otherwise, the user cannot exit from the form.

An INPUT statement can include only one AFTER INPUT block.

The NEXT FIELD Keywords

The NEXT FIELD keywords specify the next field to which 4GL moves the screen cursor. If you do not specify a NEXT FIELD clause, then by default the cursor moves among the screen fields according to the explicit or implicit order of fields in the INPUT binding clause. The user can control movement from field to field by the Arrow keys, TAB, and RETURN. By using the NEXT FIELD keywords, however, you can explicitly position the screen cursor. You must specify one of the following options with the NEXT FIELD keywords:

Clause	Effect
NEXT FIELD NEXT	Advances the cursor to the *next* field.
NEXT FIELD PREVIOUS	Returns the cursor to the *previous* field.
NEXT FIELD *field name*	Moves the cursor to *field-name*.

For example, this NEXT FIELD clause places the cursor in the previous field:

```
NEXT FIELD PREVIOUS
```

The following INPUT statement includes a NEXT FIELD clause in an ON KEY block. If the user presses CONTROL-B when the cursor is in the **stock_num** or **manu_code** fields, then 4GL moves the cursor to **quantity** as the next field:

```
INPUT p_items.* FROM s_items.*
    ON KEY (CONTROL-B)
        IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            CALL stock_help()
            NEXT FIELD quantity
        END IF
    ...
END INPUT
```

4GL immediately positions the cursor in the form when it encounters the NEXT FIELD clause; it does not execute any statements that follow the NEXT FIELD clause in the control block. For example, 4GL cannot invoke function **qty_help( )** in the next example:

```
ON KEY (CONTROL-B, F4)
    IF INFIELD(stock_num) OR INFIELD(manufact) THEN
        CALL stock_help()
        NEXT FIELD quantity
        CALL qty_help() -- function is never called
    END IF
```

You can use the NEXT FIELD clause in any INPUT form management block. The NEXT FIELD clause typically appears in a conditional statement. In an AFTER INPUT clause, the NEXT FIELD statement *must* appear in a conditional statement; otherwise, the user cannot exit from the form. To restrict access to a field, use the NEXT FIELD statement in a BEFORE FIELD clause.

The following example demonstrates using the NEXT FIELD clause in an ON KEY control block. 4GL executes the ON KEY block if the user presses CONTROL-W. If the cursor is in the **city** field, 4GL displays `San Francisco` in the city field and `CA` in the **state** field, and then moves the cursor to the **zip-code** field.

```
ON KEY (CONTROL-W)
    IF INFIELD(city) THEN
        LET p_addr.city = "San Francisco"
        DISPLAY p_addr.city TO city
        LET p_addr.state = "CA"
        DISPLAY p_addr.state TO state
        NEXT FIELD zipcode
    END IF
```

To wrap from the last field of a form to the first field of a form, use the NEXT FIELD statement after an AFTER FIELD clause for the last field of the form. (The INPUT WRAP option of the OPTIONS statement has the same effect.)

The CONTINUE INPUT Statement

The CONTINUE INPUT statement causes 4GL to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form.

The CONTINUE INPUT statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user. It is also useful in an AFTER INPUT control block that examines the field buffers; depending on their contents, you can return the cursor to the form.

The EXIT INPUT Statement

The EXIT INPUT statement terminate input. 4GL does the following:

- Skips all statements between the EXIT INPUT and END INPUT keywords.
- Deactivates the form.
- Resumes execution at the first statement after the END INPUT keywords.

4GL ignores any statements in an AFTER INPUT control block if the EXIT INPUT statement is executed.

The END INPUT Keywords

The END INPUT keywords indicate the end of the INPUT statement. These keywords should follow the last control block. If you do not include any control blocks, then the END INPUT keywords are not required.

Using Built-In Functions and Operators

The INPUT statement supports built-in functions and operators of 4GL. (For more about these built-in 4GL functions and operators, see Chapter 4.) The following features allow you to access field buffers and keystroke buffers:

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has "touched" (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or Arrow keys) does not mark a field as touched. This operator also ignores the effect of statements that appear in the BEFORE INPUT control block. For example, you can assign values to fields in the BEFORE INPUT control block without having the fields marked as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user while in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is specified as its operand is the name of the current field.

Each field has only one field buffer, and a buffer cannot be used by two different statements simultaneously. If you plan to display more than once the same form with data entry fields, you should open a new 4GL window and open and display a second copy of the form. 4GL allocates a separate set of buffers to each form, so this avoids overwriting field buffers when more than one INPUT, INPUT ARRAY, or CONSTRUCT statement accepts input.

The following statement uses the INFIELD( ) operator to determine if the cursor is in the **stock_num** or **manu_code** fields. If the cursor is in one of these fields, 4GL calls the **stock_help( )** function and sets **quantity** as the next field:

```
INPUT p_items.* FROM s_items.*
    ON KEY (CONTROL-B)
        IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            CALL stock_help()
            NEXT FIELD quantity
        END IF
```

The INFIELD(*field*) expression returns TRUE if the current field is *field* and FALSE otherwise. Use this function for field-dependent actions when the user presses a key in the ON KEY block. In the following INPUT statement, the BEFORE FIELD clause for the **city** field displays a message advising the user to press a control key to enter the value `San Francisco` into the field:

```
INPUT p_customer.fname THRU p_customer.phone
    FROM sc_cust.* ATTRIBUTE(REVERSE)
    BEFORE FIELD city
        MESSAGE "Press CONTROL-F to enter San Francisco"
    ON KEY (CONTROL-F)
        IF INFIELD(city) THEN
            LET p_customer.city = "San Francisco"
            DISPLAY p_customer.city TO city
            LET p_customer.state = "CA"
            DISPLAY p_customer.state TO state
            NEXT FIELD zipcode
        END IF
END INPUT
```

If the user presses the CONTROL-F key while the cursor is in the **city** field, the ON KEY clause in this example changes the screen display in three ways:

1. Displays the value `San Francisco` in the **city** field.
2. Displays `CA` in the **state** field.
3. Moves the cursor to the first character position in the **zipcode** field.

Keyboard Interaction

The user of your 4GL application can use the keyboard to position the cursor during an INPUT statement. The behavior of some of the keys that can position the cursor during an INPUT statement is sensitive to what kind of field the cursor occupies:

- Whether the cursor is currently in a simple field.
- Or in a segment of a multiple-segment field.

Subsequent sections describe cursor movement in both environments.

NLS

If NLS is active, the settings in the NLS environment variables LC_MONETARY and LC_NUMERIC can change the interpretation of currency symbols and numeric and decimal separators input by the user. For example, in the French or German locale, values input by the user are expected to contain commas, not periods, as decimal separators.

Cursor Movement in Simple Fields

In a simple field, when the user presses TAB or RETURN, the cursor moves from one screen field to the next in an order based on the binding clause:

- For INPUT BY NAME, 4GL uses the order implied by the sequence of program variables specified in the binding clause.
- Otherwise, 4GL uses the order of the screen fields specified in the FROM clause of the INPUT statement.

The user can press the Arrow keys to position the screen cursor:

Arrow	Effect
[↓]	By default, moves the cursor to the next field. If you specify FIELD ORDER UNCONSTRAINED in the OPTIONS statement, this key moves the cursor to the field below the current field. If no field is below the current field and a field exists to the left of the current field, **4GL** moves the cursor to the field to the left.
[↑]	By default, moves the cursor to the previous field. If you specify the FIELD ORDER UNCONSTRAINED option of the OPTIONS statement, this key moves the cursor to the field above the current field. If no field is above the current field and a field exists to the left of the current field, 4GL moves the cursor to the field to the left.
[→]	Moves the cursor one space to the right inside a screen field, without erasing the current character. At the end of the field, **4GL** moves the cursor to the first character position of the next screen field. The [→] is equivalent to the CONTROL-L editing key.

[←]	Moves the cursor one space to the left inside a screen field without erasing the current character. At the beginning of the field, **4GL** moves the cursor to the first character position of the previous field. The [←] is equivalent to the CONTROL-H editing key.

Editing Keys

Unless a field has the NOENTRY attribute, the user can press the following keys during an INPUT statement to edit values in a screen field:

Key	Effect
CONTROL-A	Toggles between insert and typeover mode.
CONTROL-D	Deletes characters from the current cursor position to the end of the field.
CONTROL-H	Moves the cursor nondestructively one space to the left. It is equivalent to pressing [←].
CONTROL-L	Moves the cursor nondestructively one space to the right. It is equivalent to pressing [→].
CONTROL-R	Redisplays the screen.
CONTROL-X	Deletes the character beneath the cursor.

Multiple-Segment Fields

Chapter 5 explains how you can create a *multiple-segment* field (page 5-26) to display long character strings; these superficially resemble a screen array, but the successive lines are *segments* of the same field, rather than screen records.

If the data string is too long to fit in the first segment, 4GL divides it at a blank character (if possible), padding the rest of the segment on the right with blank (ASCII 32) characters, and continues the display in the next field segment. If necessary, this process is repeated until all of the segments are filled, or until the last text character is displayed (whichever happens first).

If the user inserts or deletes characters while editing a multiple-segment field, WORDWRAP attribute can "wrap" subsequent characters, as needed. Blank characters that the WORDWRAP editor uses as padding are called *editor blanks.* The COMPRESS keyword in the form specification file to prevent the storage of editor blanks in the database. Characters that the user enters, or that 4GL retrieves from the database, are called *intentional* characters.

If the cursor enters a multiple-segment field, additional features of a multiple line editor become available to the user. The user must press CONTROL-R for NEWLINE, because RETURN moves to the next field.

WORDWRAP Editing Keys

When values are entered or updated in a multiple-segment field, the user can press keys to move the screen cursor over the data, and to insert, delete, and type over the data. The cursor never pauses on editor blanks.

The WORDWRAP editor has two modes, *insert* (to add data at the cursor) and *typeover* (to replace the displayed data with entered data). Users cannot overwrite a NEWLINE. If the cursor in *typeover* mode encounters a NEWLINE character, the mode automatically changes to *insert*, "pushing" the NEWLINE character to the right. Some keystrokes behave differently in the two modes.

When it first enters a multiple-segment field, the cursor is positioned on the first character of the first field segment, and the editing mode is set to *typeover*. The cursor movement keys are as follows:

RETURN — leaves the entire multiple-segment field, and goes to the first character of the next field.

BACKSPACE *or* LEFT ARROW — move left one character, unless at the left edge of a field segment. From the beginning of the first segment, these move to the first character of the preceding field (if INPUT WRAP is in effect), or beep (if INPUT NO WRAP; see the OPTIONS statement). From the left edge of a lower field segment, these keys move to the last intentional character of the previous field segment.

RIGHT ARROW — moves right one character, unless at the rightmost intentional character in a segment. From the rightmost intentional character of the last segment, this either moves to the first character of the next field, or only beeps, depending on INPUT WRAP mode. From the last intentional character of a higher segment, this moves to the first intentional character in a lower segment.

UP ARROW — moves from the topmost segment to the first character of the preceding field. From a lower segment, this moves to the character in the same column of the next higher segment, jogging left, if required, to avoid editor blanks, or if it encounters a TAB.

DOWN ARROW — moves from the lowest segment to the first character of the next field. From a higher segment, moves to the character in

	the same column in the next lower segment, jogging left if required to avoid editor blanks, or if it encounters a TAB.
TAB	enters a TAB character, in insert mode, and moves the cursor to the next TAB stop. This can cause following text to jump right to align at a TAB stop. In typeover mode, this moves the cursor to the next TAB stop that falls on an intentional character, going to the next field segment if required.

The character keys enter data. Any following data shifts right, and words can move down to subsequent segments. This can result in characters being discarded from the final field segment. These keystrokes can also alter data:

CONTROL-A	switches between typeover and insert mode.
CONTROL-X	deletes the character under the cursor, possibly causing words to be pulled up from subsequent segments.
CONTROL-D	deletes all text from the cursor to the end of the multiple-line field (not merely to the end of the current field segment).
CONTROL-N	inserts a NEWLINE character, causing subsequent text to align at the first column of the next segment of the field, and possibly moving words down to subsequent segments. This can result in characters being discarded from the final segment of the field.

The editing keys (page 3-147) have the same effect in a multiple-segment field, except that CONTROL-H can move to the last intentional character of the previous segment of the same field, if the cursor is on the first intentional character. Also, CONTROL-L can move to the first intentional character of the next segment of the same field from the last intentional character of a segment.

Using Large Data Types

4GL displays values of large data types (BYTE or TEXT) as follows:

Field Type	Screen Display
TEXT	As much of the TEXT data as fit within the screen field.
BYTE	The string `"<BYTE value>"`. **4GL** cannot display the actual BYTE value in a screen field.

Use a simple field. (You can display part of a TEXT value in a multiple-segment field, or all of a TEXT value that is short enough to fit, but the user cannot apply the WORDWRAP editor to a TEXT value.)

If the form specification file assigns an appropriate attribute to a BYTE or TEXT field, the user can invoke an external program by pressing the exclamation (!) key when the cursor is in the field. This external program is typically an editor to allows the user to edit character (TEXT) or graphic (BYTE) data. To implement this feature, specify the PROGRAM attribute as part of the field description in the form specification file, identifying the external program to execute. (For more information on using the PROGRAM attribute, see the description of that field attribute in Chapter 5.)

The external program takes over the entire screen. Any key sequence that you have specified in the ON KEY clause is ignored by the external program. When the external program terminates, 4GL does the following:

1. Restores the screen to its state before the external program began.
2. Resumes the INPUT statement at the BYTE or TEXT field.
3. Reactivates any key sequences specified in the ON KEY clause.

Completing the INPUT Statement

The following actions can terminate the INPUT statement:

- The user chooses one of the following keys:
  - Accept, Interrupt, or Quit
  - The RETURN or TAB key from the last field (and INPUT WRAP is not currently set by the OPTIONS statement)
- 4GL executes the EXIT INPUT statement.

By default, the Accept, Interrupt, or Quit keys terminate the INPUT statement. Each of these actions also deactivates the form. (But pressing the Interrupt or Quit key can immediately terminate the program, unless the program also includes the DEFER INTERRUPT and DEFER QUIT statements.)

The user must choose Accept explicitly to complete the INPUT statement under the following conditions:

- INPUT WRAP is specified in the OPTIONS statement.
- An AFTER FIELD block for the last field includes a NEXT FIELD clause.

If 4GL previously executed a DEFER INTERRUPT statement in the program, the Interrupt key causes 4GL to do the following:

- Set the global variable **int_flag** to a nonzero value.
- Terminate the INPUT statement, but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, a Quit signal causes 4GL to do the following:

- Set the global variable **quit_flag** to a nonzero value.
- Terminate the INPUT statement, but not the 4GL program.

Executing Control Blocks When INPUT Terminates

When INPUT terminates, these blocks are executed in the order indicated:

1. The AFTER FIELD clause for the current field.
2. The AFTER INPUT clause.

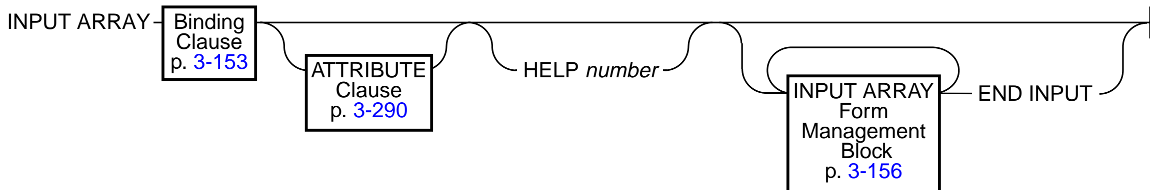
If INPUT terminates by an EXIT INPUT statement, or by pressing the Interrupt or Quit keys, 4GL does not execute any of these clauses. If a NEXT FIELD statement appears in one of these clauses, 4GL places the cursor in the specified field and returns control to the user.

References

DEFER, DISPLAY ARRAY, INPUT ARRAY, OPEN WINDOW, OPTIONS

INPUT ARRAY

The INPUT ARRAY statement supports data entry by users into a screen array, and stores the entered data in a program array of records.



number is a literal integer (page 3-340) to specify a Help message number.

Usage

The INPUT ARRAY statement assigns to variables in one or more program records the values that the user enters into the fields of a screen array. This statement can include statement blocks to be executed under conditions that you specify, such as screen cursor movement, or other user actions. To use the INPUT ARRAY statement, you must do the following:

1. Specify a screen array in the form specification file, and compile the form.
2. Declare an ARRAY OF RECORD with the DEFINE statement.
3. Open and display the screen form in either of the following ways:
   - The OPEN FORM and DISPLAY FORM statements.
   - An OPEN WINDOW statement with the WITH FORM clause.
4. Use the INPUT ARRAY statement to assign values to the *program array* from data that the user enters into fields of the *screen array.*

When the INPUT ARRAY statement is encountered, 4GL does the following:

1. Displays any default values in the screen fields, unless you specify the WITHOUT DEFAULTS keywords (as described on page 3-131.)
2. Moves the cursor to the first field and waits for input from the user.
3. Assigns the user-entered field value to a corresponding program variable when the user moves the cursor from the field or presses the Accept key.

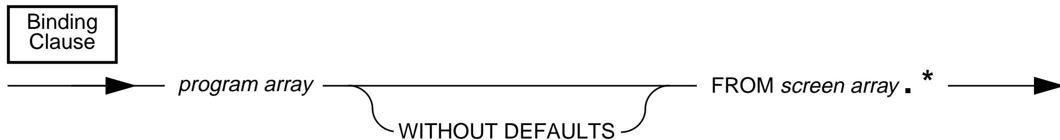
The INPUT ARRAY statement activates the *current form* (the form that was most recently displayed, or the form in the current 4GL window). When the INPUT ARRAY statement completes execution, the form is deactivated. After the user presses Accept, the INSERT statement of SQL can insert the values of the program variables into the appropriate database tables.

The following topics are described in this section:

Topic	Page
The Binding Clause	3-153
The ATTRIBUTE Clause	3-155
The HELP Clause	3-133
The INPUT ARRAY Form Management Blocks	3-158
The EXIT INPUT Statement	3-169
The END INPUT Keywords	3-169
Using Built-In Functions and Operators	3-170
Keyboard Interaction	3-172
Using Large Data Types	3-149
Completing the INPUT ARRAY Statement	3-174

The Binding Clause

The *binding clause* temporarily associates the member variables in an array of program records with fields in the member records of a screen array, so the 4GL program can manipulate values that the user enters in the screen array:



program array is the name of an array of program records.

screen array is the name of an array of screen records.

You must declare the *program array* in your program, and the *screen array* in the form specification file.

The Correspondence of Variables and Fields

The form can include other fields that are not part of the specified screen array, but the number of member variables in each record of *program array* must equal the number of fields in each row of *screen array.* Each variable must be of the same (or a compatible) data type as the corresponding screen field. When the user enters data, 4GL checks the entered value against the data type of the variable, not the data type of the screen field

The member variables of the records in *program array* can be of any 4GL data type. If a variable is declared LIKE a SERIAL column, however, 4GL does not allow the screen cursor to stop in the field. (Values in SERIAL columns are maintained by the database engine, not by 4GL.)

The number of screen records in *screen array* determines how many rows the form can display at one time. The size of *record array* determines how many RECORD variables your program can store. If the size of a program array exceeds the size of its screen array, users can press the Next Page or Previous Page keys (page 3-172) to scroll through the screen array.

The default order in which the screen cursor moves from field to field in the screen array is determined by the declared order of the corresponding member variables, beginning in the first screen record. (See also the NEXT FIELD keywords on page 3-141, and the WRAP and FIELD ORDER options of the OPTIONS statement, as described on page 3-232.)

Displaying Default Values

If you omit the WITHOUT DEFAULTS keywords, 4GL displays default values from the program array when the form is activated. 4GL determines the default values in the following way, in descending order of precedence:

1. The DEFAULT attribute (from the form specification file).
2. The DEFAULT column value (from the **syscolval** table).

4GL assigns NULL values to all variables for which no default is set. But if you include the WITHOUT NULL INPUT option in the DATABASE section of the form specification file, then 4GL assigns the following default values:

Field Type	Default
character	blank (= ASCII 32)
number	0
INTERVAL	0
MONEY	$0.00
DATE	12/31/1899
DATETIME	1899-12-31 23:59:59.99999

The WITHOUT DEFAULTS Keywords

If you specify the WITHOUT DEFAULTS option, however, the screen displays *current* values of the variables when the INPUT ARRAY statement begins. This option is available with both the BY NAME and the FROM binding clauses.

To display initialized values, rather than defaults, you can do the following:

1. Initialize the variables with whatever values you want to display.
2. Call the built-in SET_COUNT( ) function to tell 4GL how many rows are currently stored in the program array.

3. Specify INPUT ARRAY . . . WITHOUT DEFAULTS to display current values of the program array, and to allow the user to change those records.

The WITHOUT DEFAULTS clause is useful when you want the user to be able to make changes to existing rows of the database. You can display the existing database values on the screen before the user begins editing the data. The FIELD_TOUCHED( ) operator can help you to determine which fields have been altered, and that therefore require updates to the database. (This operator is described briefly on page 3-170, and in detail on page 4-64.)

The ATTRIBUTE Clause

For the syntax of the ATTRIBUTE clause, see page 3-290. This section describes the use of the ATTRIBUTE clause within an INPUT ARRAY statement.

If you specify form attributes with the INPUT ARRAY statement, the new attributes apply only during the current activation of the form. When actions of the user deactivate the form, the form reverts to its previous attributes. The following INPUT statement assigns the RED and REVERSE attributes:

```
INPUT ARRAY p_addr FROM sc_addr.* ATTRIBUTE (RED, REVERSE)
```

This statement assigns the WHITE attribute:

```
INPUT ARRAY p_items FROM sc_items.* ATTRIBUTE (WHITE)
```

The ATTRIBUTE clause overrides any default display attributes specified in an OPTIONS or OPEN WINDOW statement for these fields. It also suppresses any default attributes specified in the **syscolatt** table of the **upscol** utility.

The HELP Clause

The HELP clause specifies the *number* of a Help message to display if the user presses the Help key while the screen cursor is in any field of the specified screen array. The default Help key is CONTROL-W, but you can assign a different key as the Help key by using the HELP KEY clause of the OPTIONS statement.

The following program fragment specifies Help message `311` if the user requests Help from any field in the `s_items` screen array:

```
INPUT ARRAY p_items FROM s_items.*
    HELP 311
```

You create Help messages in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement (page 3-234). A run-time error occurs in the following situations:

- 4GL cannot open the Help file.
- You specify a *number* that is not in the Help file.
- You specify a *number* outside the range from -32,767 to 32,767.

The Help message specified in your HELP clause applies to the entire INPUT ARRAY statement. To override this with field-level Help messages, specify an ON KEY block (page 3-137) that invokes the INFIELD( ) operator and SHOW_HELP( ) function, as described in Chapter 4. If you do this, the messages must be displayed in a 4GL window within the 4GL screen, rather than in the separate Help window.

The INPUT ARRAY Form Management Blocks

Each INPUT ARRAY form management block includes a *statement block* of at least one statement, and an *activation clause* that specifies when to execute the *statement block*. An INPUT ARRAY form management block can specify any of the following:

- The statements to execute before or after visiting specific screen fields.
- The statements to execute when the user presses a key sequence.
- The statements to execute before or after the INPUT ARRAY statement.
- The next field to which to move the screen cursor.
- When to terminate execution of the INPUT ARRAY statement.

The *activation clause* can specify any one of the following:

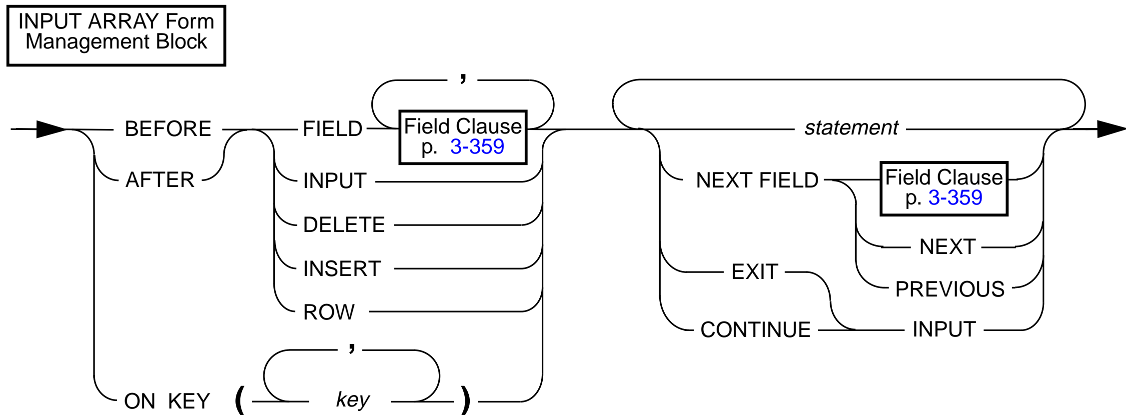
- Pre- and post-INPUT actions (the BEFORE or AFTER INPUT clause)
- Pre- and post-INSERT actions (the BEFORE or AFTER INSERT clause)
- Pre- and post-DELETE actions (the BEFORE or AFTER DELETE clause)
- Keyboard sequence conditions (the ON KEY clause)
- Cursor movement conditions (the BEFORE or AFTER FIELD clause, and the BEFORE or AFTER ROW clause)

The *statement block* can include any SQL or 4GL statements, as well as:

- Cursor movement instructions (the NEXT FIELD clause)
- Termination of the INPUT ARRAY statement (the EXIT INPUT statement)
- Returning control to the user, without terminating the INPUT ARRAY statement (the CONTINUE INPUT statement)

If you include one or more form management blocks, the END INPUT keywords must terminate the INPUT ARRAY statement. If no form management block is included, 4GL waits while the user enters values into the fields. When the user presses the Accept key, the INPUT ARRAY statement terminates.

The *activation clause* and the *statement block* correspond respectively to the left-hand and right-hand elements in the following syntax diagram:

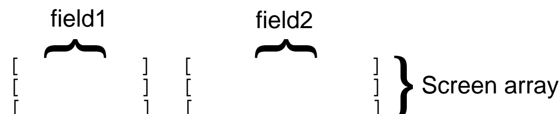


keyname is one or more keywords to specify physical or logical keys. For details, see "The ON KEY Block" on page 3-137.

statement is an SQL or other 4GL statement.

After BEFORE FIELD, AFTER FIELD, or NEXT FIELD, the Field clause specifies a field that the binding clause referenced implicitly (in the BY NAME clause, or as *record*.* or *array* [*line*].*) or explicitly. You can qualify a field name by a *table* reference, or the name of a *screen record* or a *screen array* or *array* [*line*].

The BEFORE FIELD *screen-array* or AFTER FIELD *screen-array* activation clause applies to the entire screen-array. BEFORE FIELD *screen-array. field* or AFTER FIELD *screen-array. field* applies to the specified field in the screen array, as in the following example, which represents part of a screen form:



If you specify BEFORE FIELD *screen array.field1,* 4GL executes the statement block if the cursor moves into the *field1* field of any screen record of *screen array,* but not after movement to a *field2* field. You can specify BEFORE FIELD *screen array* if you want the statement block to be executed if the cursor enters any field of *screen array.*

If you include a form management block, 4GL executes or ignores the statements in a form management block, depending on the following:

- Whether you specify the BEFORE INPUT or AFTER INPUT keywords.
- The *fields* to which and from which the user moves the screen cursor.
- The *keys* that the user presses.

4GL deactivates the form while executing statements in a form management block. After executing the statements, 4GL reactivates the form, allowing the user to continue entering or modifying the data values in fields.

The Precedence of Form Management Blocks

This is the order in which 4GL executes the statements in control blocks:

1. BEFORE INPUT
2. BEFORE ROW
3. BEFORE INSERT, BEFORE DELETE
4. BEFORE FIELD *screen array*
5. BEFORE FIELD *screen array. field*
6. ON KEY
7. AFTER FIELD *screen array. field*
8. AFTER FIELD *screen array*
9. AFTER INSERT, AFTER DELETE
10. AFTER ROW
11. AFTER INPUT

You can list these blocks in any order. If you develop some consistent ordering, however, your code may be easier to read.

Within these blocks, you can include the NEXT FIELD keywords (page 3-141) and EXIT INPUT statement (page 3-143), as well as most 4GL and SQL statements. You cannot specify a CONSTRUCT, PROMPT, INPUT, nor INPUT ARRAY statement, but you can invoke a function that executes one or more of these statements.

The activation clauses of INPUT ARRAY form management blocks are described in their order of execution by 4GL. Descriptions of NEXT FIELD and EXIT INPUT follow the discussions of these activation clauses. No subsequent control block statements are executed if EXIT INPUT executes.

The BEFORE INPUT Block

You can use the BEFORE INPUT block to display messages on how to use the INPUT ARRAY statement. For example, the following statement fragment displays a message informing the user how to enter data into the table:

```
INPUT ARRAY p_customer FROM s_customer.*
    BEFORE INPUT
        DISPLAY "Press ESC to enter data" AT 1,1
```

4GL executes the BEFORE INPUT block after displaying the default values in the fields and before letting the user enter any values. (If you included the WITHOUT DEFAULTS clause, 4GL displays the current values of the variables, not the default values, before executing the BEFORE INPUT block.)

The following program fragment uses the DISPLAY statement in the BEFORE INPUT block to populate the fields of a single screen array:

1. Call SET_COUNT(1) to initialize the array with one non-default record.
2. Include the WITHOUT DEFAULTS block in the INPUT ARRAY statement.
3. Within the BEFORE INPUT block, use LET statements to assign values to the variables. Then use DISPLAY to display the variable to the screen:

```
CALL SET_COUNT(1)
INPUT ARRAY p_items WITHOUT DEFAULTS FROM s_items.*
    BEFORE INPUT
        LET pa_curr = ARR_CURR()
        LET s_curr = SCR_LINE()
        LET p_items[pa_curr].stock_num = 2
        DISPLAY p_items[pa_curr].stock_num TO
            s_items[s_curr].stock_num
        NEXT FIELD manu_code
END INPUT
```

An INPUT ARRAY statement can include only one BEFORE INPUT block. You cannot include the FIELD_TOUCHED( ) operator in the BEFORE INPUT block.

The BEFORE ROW Block

Here "ROW" means a screen record; it need not be linked to a database row. The INPUT ARRAY statement can include no more than one BEFORE ROW block. 4GL executes the BEFORE ROW block statements in the following cases:

- The cursor moves into a new line of the screen form.
- An INSERT statement fails because of lack of space.
- An INSERT statement is terminated by the Interrupt or Quit key.
- The user presses the Delete key.

The BEFORE DELETE Block

This statement block is executed after the user presses the Delete key while the cursor is in a screen array, but before 4GL actually deletes the record. An INPUT ARRAY statement can include only one BEFORE DELETE block.

If you want to prevent the record from being deleted (for example, if some Boolean condition is not satisfied), specify EXIT INPUT, rather than CONTINUE INPUT, within the BEFORE DELETE block.

The BEFORE INSERT Block

Statements in the BEFORE INSERT block are executed in these cases:

- When the user begins entering new records into the array.
- After the user presses the Insert key to insert a new record between existing records of a screen array, but before the record is added to the array.
- When the user moves the cursor to a blank record at the end of an array.

4GL executes the statements in this block before the user enters data for each successive screen record that the Insert key creates.

The following BEFORE INSERT block calls the **get_item_num( )** function before inserting a new empty record into the screen array:

```
BEFORE INSERT
    CALL get_item_num()
```

An INPUT ARRAY statement can include only one BEFORE INSERT block.

The BEFORE FIELD Block

This statement block is executed whenever the screen cursor moves into the specified field, but before the user enters a value. You can specify no more than one BEFORE FIELD block for each field.

The following program fragment defines two BEFORE FIELD blocks. When the cursor enters the **fname** or **lname** field, 4GL displays a message:

```
BEFORE  FIELD  fname
    MESSAGE  "Enter  first  name  of  customer"
BEFORE  FIELD  lname
    MESSAGE  "Enter  last  name  of  customer"
```

You can use a NEXT FIELD clause within a BEFORE FIELD block to restrict access to a field. You can also use a DISPLAY statement within a BEFORE FIELD block to display a default value in a field.

The following statement fragment causes 4GL to prompt the user for input when the cursor is in the **stock_num**, **manu_code**, or **quantity** fields:

```
INPUT ARRAY p_items FROM s_items.*
    BEFORE FIELD stock_num
        MESSAGE "Enter a stock number."
    BEFORE FIELD manu_code
        MESSAGE "Enter the code for a manufacturer."
    BEFORE FIELD quantity
        MESSAGE "Enter a quantity."
    ...
END INPUT
```

The ON KEY Block

Statements in the ON KEY block are executed if the user presses some key that you specify by these keywords (in lowercase or uppercase letters):

ACCEPT	HELP	NEXT *or*	RETURN
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS *or*	TAB
ESC *or* ESCAPE	LEFT	PREVPAGE	UP

F1 *through* F64

CONTROL-*char* (except A, D, H, I, J, L, M, R, or X)

This statement defines an ON KEY block for the CONTROL-B key. Whenever the user presses the CONTROL-B key, 4GL determines if the screen cursor is in the **stock_num** or **manu_code** fields. If it is in either one of these fields, then 4GL calls the **stock_help( )** function and sets **quantity** as the next field.

```
INPUT ARRAY p_items FROM s_items.*
    ON KEY (CONTROL-B)
        IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            CALL stock_help()
            NEXT FIELD quantity
        END IF
```

The following ON KEY block displays a Help message. The BEFORE INPUT clause informs the user how to access Help:

```
BEFORE INPUT
    DISPLAY "Press CTRL-W for help"
ON KEY (CONTROL-W, CONTROL-F)
    CALL customer_help()
```

Some keys require special consideration if specified in an ON KEY block:

Key	**Special Considerations**
ESC or ESCAPE	You must specify another key as the Accept key in the OPTIONS statement, because this is the default Accept key.
INTERRUPT	You must execute a DEFER INTERRUPT statement. If the user presses the Interrupt key under these conditions, **4GL** executes the statements in the ON KEY block and sets **int_flag** to nonzero, but does not terminate the INPUT statement. **4GL** also executes the statements in this ON KEY block if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, **4GL** sets **quit_flag** to non-zero.
CONTROL-W	You must specify another key as the Insert key in the OPTIONS statement, because CONTROL-W is the default Insert key.
F2	You must specify another key as the Delete key in the OPTIONS statement, because F2 is the default Delete key.
F3	You must specify another key as the Next Page key in the OPTIONS statement, because F3 is the default Next Page key.
F4	You must specify another key as the Previous Page key in the OPTIONS statement, because F4 is the default for that key.

CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these keys for field editing.
I, J, and M	The regular meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is lost to the user. Instead, the key is trapped by **4GL** and used to activate the ON KEY block. For example, if CONTROL-M appears in an ON KEY block, the user cannot press RETURN to advance the cursor to the next field. If you include one of these keys in an ON KEY block, be careful to restrict the scope of the block to specific fields.

You may not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

If you use the OPTIONS statement to redefine the Accept or Help keys, the keys assigned to these sequences cannot be used in an ON KEY clause. For example, if you redefine the Accept key by using the following statement, you should not define an ON KEY block for the key sequence CONTROL-B:

```
OPTIONS ACCEPT KEY (CONTROL-B)
```

When the user presses CONTROL-B, 4GL will always perform the Accept key function, regardless of the presence of an ON KEY (CONTROL-B) block.

If the user activates an ON KEY block while entering data in a field, 4GL takes the following actions:

1. Suspends input to the current field.
2. Preserves the input buffer that contains the characters the user has typed.
3. Executes the statements in the current ON KEY block.
4. Restores the input buffer for the current screen field.
5. Resumes input in the same field, with the screen cursor at the end of the buffered list of characters.

You can change this default behavior by performing the following tasks in the ON KEY block:

- Resuming input in another field by using the NEXT FIELD statement.
- Changing the input buffer value for the current field by assigning a new value to the corresponding variable, and then displaying this value.

You can also use this block to provide *accelerator keys* for common functions, such as saving and deleting. The INFIELD( ) operator can control field-specific responses in the action for an ON KEY clause. You can implement field-level Help by using the INFIELD( ) operator and SHOWHELP( ) function.

The AFTER FIELD Block

4GL executes the statements in the AFTER FIELD block associated with a field every time the cursor leaves the field. Any of the following keys can cause the cursor to leave the field.

- Any arrow key
- RETURN or TAB key
- Accept key
- Interrupt or Quit key (if a supporting DEFER statement was executed)

You can specify only one AFTER FIELD block for each field.

This AFTER FIELD block checks if the **stock_num** and **manu_code** fields contain values. If they contain values, 4GL calls the **get_item( )** function:

```
AFTER FIELD stock_num, manu_code
    LET pa_curr = ARR_CURR()
    IF p_items[pa_curr].stock_num IS NOT NULL
        AND p_items[pa_curr].manu_code IS NOT NULL THEN
            CALL get_item()
            IF p_items[pa_curr].quantity IS NOT NULL THEN
                CALL get_total()
            END IF
    END IF
```

The following statement makes sure that the user enters an address line:

```
INPUT ARRAY p_addr FROM sc_addr.*
    AFTER FIELD address1
        IF p_addr.address1 IS NULL THEN
            NEXT FIELD address1
        END IF
END INPUT
```

The user terminates the INPUT ARRAY statement by pressing the Accept key when the screen cursor is in any field, or by pressing the RETURN or TAB key after the *last* field. You can use the AFTER FIELD block on the last field to override this default termination. (Including the INPUT WRAP in the OPTIONS statement produces the same effect.)

When the NEXT FIELD keywords appear in an AFTER FIELD block, the cursor moves to in the specified field. If an AFTER FIELD block appears for each field, and NEXT FIELD keywords are in each block, the user cannot leave the form.

The AFTER INSERT Block

This has no effect unless the BY NAME or FROM clause references a screen array. 4GL executes the AFTER INSERT block after the user inserts a record into the screen array. A user inserts a record by doing the following:

1. Entering information in all the required fields of the current record.
2. Moving the cursor out of the last input field by using one of these keys:
   - Any arrow key
   - RETURN or TAB key
   - Accept key
   - Home or End key

Notice that the Insert key does *not* by itself activate the AFTER INSERT block; the user must also move the cursor from the newly inserted record.

The following AFTER INSERT block calls the **renum_items( )** function after the user inserts a new blank screen record into the **items** screen array:

```
AFTER INSERT OF items
    CALL renum_items()
```

An INPUT statement can include only one AFTER INSERT block.

The AFTER DELETE Block

4GL executes the AFTER DELETE block after the user deletes the values from a screen record by using the Delete key. If this block is present, 4GL takes the following actions when the user presses the Delete key:

1. Deletes the record from the screen array.
2. Executes the statements in the AFTER DELETE block.
3. Executes the statements in the AFTER ROW block, if one is specified.

The user must also press the Accept key to make corresponding changes to the variables in the array of program records. The following AFTER DELETE block calls the **renum_items( )** function:

```
AFTER DELETE OF items
    CALL renum_items()
```

An INPUT statement can include only one AFTER DELETE block.

The AFTER ROW Block

Here "ROW" means a screen record; this need not be linked to a database row. 4GL executes the statements in the AFTER ROW block in these cases:

- When the cursor leaves the current row by using one of these keys:
  - Any Arrow key
  - The RETURN or TAB key
  - The Accept key
  - The Interrupt key (if DEFER INTERRUPT was also executed)
- When a new screen record is inserted by the Insert key.

The INPUT ARRAY statement can specify only one AFTER ROW block. If you specify both an AFTER ROW and an AFTER INSERT block, 4GL executes the AFTER ROW block immediately after executing the AFTER INSERT block.

The following AFTER ROW block calls the **order_total( )** function after the screen cursor leaves a row, and the row is inserted:

```
AFTER ROW
    CALL order_total()
```

If you include a NEXT FIELD statement in an AFTER ROW block, **4GL** moves the cursor to the next field of the next row, not to the row which the cursor has just left.

The AFTER INPUT Block

The statements in the AFTER INPUT block are executed when the user terminates the INPUT ARRAY statement without terminating the 4GL program.

By using the GET_FLDBUF( ) or FIELD_TOUCHED( ) built-in operators within the AFTER INPUT block, you can use the AFTER INPUT block to validate, save, or alter values that the user entered. (For the use of these operators in an

INPUT ARRAY statement, see "Using Built-In Functions and Operators" on page 3-170.) The following example uses this block to require that a first name be specified for any customers with the last name `Smith`:

```
CALL SET_COUNT(1)
INPUT ARRAY p_customer FROM sc_customer.*
    AFTER INPUT
        IF p_customer.lname="Smith" THEN
            IF NOT FIELD_TOUCHED(p_customer.fname) THEN
                CALL mess("You must enter a first name.")
                NEXT FIELD fname
            END IF
        END IF
    END INPUT
```

4GL executes the AFTER INPUT block only when the INPUT ARRAY statement is terminated by the user pressing one of the following keys:

- The Accept key.
- The Interrupt key (if the DEFER INTERRUPT statement has executed).
- The Quit key (if the DEFER QUIT statement has executed).

The AFTER INPUT clause is not executed in the following situations:

- The user presses the Interrupt or Quit key and the DEFER INTERRUPT or DEFER QUIT statement, respectively, has not executed. In either case, the program terminates immediately.
- The EXIT INPUT statement terminates the INPUT ARRAY statement.

You can place the NEXT FIELD clause in this block to return the cursor to the form. If you place a NEXT FIELD clause in the AFTER INPUT block, use it in a conditional statement. Otherwise, the user cannot exit from the form.

An INPUT ARRAY statement can include only one AFTER INPUT block.

The NEXT FIELD Keywords

The NEXT FIELD keywords specify the next field to which 4GL moves the screen cursor. If you omit this clause, then by default the cursor moves among the screen fields according to the explicit or implicit order of fields in the INPUT ARRAY binding clause. The user can control movement from field to field by the Arrow keys, TAB, and RETURN. By using the NEXT FIELD keywords, however, you can explicitly position the screen cursor. You must specify one of the following options with the NEXT FIELD keywords:

Clause	Effect
NEXT FIELD NEXT	Advances the screen cursor to the *next* field.
NEXT FIELD PREVIOUS	Returns the screen cursor to the *previous* field.
NEXT FIELD *field name*	Moves the screen cursor to *field-name*.

For example, this NEXT FIELD clause places the cursor in the previous field:

```
NEXT FIELD PREVIOUS
```

The following INPUT ARRAY statement includes a NEXT FIELD clause in an ON KEY block. If the user presses CONTROL-B when the screen cursor is in the **stock_num** or **manu_code** fields, 4GL sets **quantity** as the next field:

```
INPUT ARRAY p_items FROM s_items.*
    ON KEY (CONTROL-B)
        IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            CALL stock_help()
            NEXT FIELD quantity
        END IF
    ...
END INPUT
```

4GL immediately positions the screen cursor in the form when it encounters the NEXT FIELD clause; it does not execute any statements that follow the NEXT FIELD clause in the control block. For example, 4GL does not invoke the **qty_help( )** function in the next example:

```
ON KEY (CONTROL-B, F4)
    IF INFIELD(stock_num) OR infield(manufact) THEN
        CALL stock_help()
        NEXT FIELD quantity
        CALL qty_help() -- function is never called
    END IF
```

You can use the NEXT FIELD clause in any INPUT ARRAY form management block. The NEXT FIELD clause typically appears in a conditional statement. In an AFTER INPUT clause, the NEXT FIELD statement *must* appear in a conditional statement; otherwise, the user cannot exit from the form. To restrict access to a field, use the NEXT FIELD statement in a BEFORE FIELD clause.

The following example demonstrates using the NEXT FIELD clause in an ON KEY control block. 4GL executes the ON KEY block if the user presses CONTROL-W.

If the cursor is in the **city** field, 4GL displays `San Francisco` in the **city** field and `CA` in the **state** field, and then moves the cursor to the **zipcode** field.

```
ON KEY (CONTROL-W)
    IF INFIELD(city) THEN
        LET p_addr.city = "San Francisco"
        DISPLAY p_addr.city TO city
        LET p_addr.state = "CA"
        DISPLAY p_addr.state TO state
        NEXT FIELD zipcode
    END IF
```

To wrap from the last field of a form to the first field of a form, use the NEXT FIELD statement after an AFTER FIELD clause for the last field of the form. (The INPUT WRAP option of the OPTIONS statement has the same effect.)

The CONTINUE INPUT Statement

The CONTINUE INPUT statement causes 4GL to skip all subsequent statements in the current control block. The screen cursor returns to the most recently occupied field in the current form.

The CONTINUE INPUT statement is useful when program control is nested within multiple conditional statements, and you want to return control to the user. It is also useful in an AFTER INPUT control block that examines the field buffers; depending on their contents, you can return the cursor to the form.

The EXIT INPUT Statement

The EXIT INPUT statement terminates input. 4GL does the following:

- Skips all statements between the EXIT INPUT and END INPUT keywords.
- Deactivates the form.
- Resumes execution at the first statement after the END INPUT keywords.

4GL ignores any statements in an AFTER INPUT control block if the EXIT INPUT ARRAY statement is executed.

The END INPUT Keywords

The END INPUT keywords indicate the end of the INPUT ARRAY statement. These keywords should follow the last control block. If you do not include any control blocks, then the END INPUT keywords are not required.

Using Built-In Functions and Operators

INPUT ARRAY supports various built-in functions and operators of 4GL. (For more about the built-in 4GL functions and operators, see Chapter 4.) The following features allow you to access field buffers and keystroke buffers:

Feature	Description
FIELD_TOUCHED( )	Returns TRUE when the user has "touched" (made a change to) a screen field whose name is passed as an operand. Moving the screen cursor through a field (with the RETURN, TAB, or Arrow keys) does not mark a field as touched. This operator also ignores the effect of statements that appear in the BEFORE INPUT control block. For example, you can assign values to fields in the BEFORE INPUT control block without having the fields marked as touched.
GET_FLDBUF( )	Returns the character values of the contents of one or more fields in the currently active form.
FGL_LASTKEY( )	Returns an INTEGER value corresponding to the most recent keystroke executed by the user while in the screen form.
INFIELD( )	Returns TRUE if the name of the field that is specified as its operand is the name of the current field.

Each field has only one field buffer; two statements cannot use a buffer simultaneously. To display more than once the same form with data entry fields, open a new 4GL window, and open and display a second copy of the form. (4GL allocates a separate set of buffers to each form, so this avoids overwriting buffers when two or more concurrent statements accept input.)

The following statement uses the INFIELD( ) operator to determine if the cursor is in the **stock_num** or **manu_code** fields. If the cursor is in one of these fields, 4GL calls the **stock_help( )** function and sets **quantity** as the next field:

```
INPUT ARRAY p_items FROM s_items.*
    ON KEY (CONTROL-B)
        IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
            CALL stock_help()
            NEXT FIELD quantity
        END IF
```

The INFIELD(*field*) expression returns TRUE if the current field is *field* and FALSE otherwise. You can use this function to control field-dependent actions when the user presses a specified key in the ON KEY control block. In the

following INPUT ARRAY statement, the BEFORE FIELD control block for the **city** field displays a message identifying a key that the user can press to enter the character string "`San Francisco`" into the field:

```
INPUT ARRAY pr_customer FROM sc_cust.* ATTRIBUTE(REVERSE)
    BEFORE FIELD city
        MESSAGE "Press CTRL-F for default city, San Francisco"
    ON KEY (CONTROL-F)
        IF INFIELD(city) THEN
            LET p_customer.city = "San Francisco"
            DISPLAY p_customer.city TO city
            LET p_customer.state = "CA"
            DISPLAY p_customer.state TO state
            NEXT FIELD zipcode
        END IF
END INPUT
```

If the user presses the CONTROL-F key while the cursor is in the **city** field, the ON KEY clause in this example changes the screen display in three ways:

1. Displays the value `San Francisco` in the **city** field.
2. Displays `CA` in the **state** field.
3. Moves the cursor to the first character position in the **zipcode** field.

You can use the following built-in functions to keep track of the relative states of the screen cursor, the program array, and the screen array:

Function	**Description**
ARR_CURR( )	Returns the number of the *current record* of the program array. This corresponds to the position of the screen cursor at the beginning of the BEFORE or AFTER ROW control block, rather than the line to which the screen cursor moves after execution of the block.
ARR_COUNT( )	Returns the current number of records in the program array.
SCR_LINE( )	Returns the number of the current line within the screen array. This is the line containing the screen cursor at the beginning of the BEFORE ROW or AFTER ROW control block, rather than the line to which the screen cursor moves after execution of the block. This number can be different from the value returned by ARR_CURR( ) if the program array is larger than the screen array.
SET_COUNT( )	Takes the number of records currently in the program array as an argument, and sets the initial value of ARR_COUNT( ).

Keyboard Interaction

The user of your 4GL application can use the keyboard to position the cursor during an INPUT ARRAY statement, to scroll the screen array, and to edit data in screen records.

By default, the user can move the cursor within a screen array and scroll the displayed rows by clicking the Arrow, Page Up, or Page Down keys and the F3 and F4 function keys. The following table describes these keys:

Key	Effect
[→]	Moves the cursor one space to the right inside a screen field without erasing the current character. At the end of the field, **4GL** moves the cursor to the first character position of the next screen field. The [→] is equivalent to the CONTROL-L editing key.
[←]	Moves the cursor one character position to the left inside a screen field without erasing the current character. At the end of the field, **4GL** moves the cursor to the first character position of the previous screen field. The [←] is equivalent to the CONTROL-H editing key.
[↓]	Moves the cursor to the same display field one line down on the screen. If the cursor was on the last line of the screen array before [↓] was used, **4GL** scrolls the program array data up one line. If the last program array record is already on the last line of the screen array, [↓] generates a message indicating that there are "no more rows in that direction."
[↑]	Moves the cursor to the same field one line up on the screen. If the cursor were on the first line of the screen array, **4GL** scrolls the program array data down one line. If the first program array record is already on the first screen array line, [↑] generates a message indicating that there are "no more rows in that direction."
F3	Scrolls the display to the next full page of program records. The NEXT KEY clause of the OPTIONS statement can reset this key.
F4	Scrolls the display to the previous full page of program records. The PREVIOUS KEY clause of the OPTIONS statement can reset this key.

Clearing Reserved Lines

When moving the cursor to a new field of an array, the INPUT ARRAY statement clears the Comment line and the Error line. The Comment line displays text defined with the COMMENTS attribute in the form specification file. The Error line displays system error messages and ERROR statement text.

Inserting and Deleting Records from an Array

The user can insert and delete records within the screen arrays by using the CONTROL-W key (the default Insert key) and F2 key (the default Delete key):

Key	Effect
CONTROL-W	Inserts a new blank screen record into the screen array at the line below the cursor. Any displayed values in lower records move down by one line, and the cursor moves to the beginning of the first field of the new blank record. This key is not needed to insert rows at the end of the screen array. If the user attempts to insert more rows than the declared size of the program array, then **4GL** displays a message that the array is full. The OPTIONS statement can specify a different physical key as the Insert key.
F2	Deletes the current record from the screen array. **4GL** adjusts any subsequent rows to fill the gap. The OPTIONS statement can specify a different physical key as the Delete key.

Pressing the Accept key makes corresponding changes in the program array. You can then use the **arr_count( )** function to determine how many records (possibly including blank records) remain in the program array after the user has pressed the Insert or Delete keys and the Accept key.

See also the BEFORE DELETE (3-160), BEFORE INSERT (3-160), AFTER INSERT (3-165) and AFTER DELETE (3-165) control blocks.

Editing Keys

Unless a field has the NOENTRY attribute, the user can press the following keys during an INPUT ARRAY statement to edit values in a field:

Key	Effect
CONTROL-A	Toggles between insert and typeover mode.
CONTROL-D	Deletes characters from the current cursor position to the end of the field.
CONTROL-H	Moves the cursor nondestructively one space to the left. It is equivalent to pressing [←].
CONTROL-L	Moves the cursor nondestructively one space to the right. It is equivalent to pressing [→].
CONTROL-R	Redisplays the screen.
CONTROL-X	Deletes the character beneath the cursor.

Using Large Data Types

Within a field of a screen array, 4GL displays any value of a large data type (BYTE or TEXT) in the following way:

Field Type	Screen Display
TEXT	As much of the TEXT data as can fit in the screen field.
BYTE	The string `<BYTE value>`. **4GL** cannot display the actual BYTE value in a screen field.

If the form specification file assigns an appropriate attribute to a BYTE or TEXT field, the user can invoke an external program by pressing the exclamation (!) key when the cursor is in the field. This external program is typically an editor that allows the user to edit large string (TEXT) or graphic (BYTE) data. To implement this feature, specify the PROGRAM attribute as part of the field description in the form specification file, identifying the external program to execute. (For more information on using the PROGRAM attribute, see the description of that field attribute in Chapter 5.)

The external program takes over the entire screen. Any key sequence that you have specified in the ON KEY clause is ignored by the external program. When the external program terminates, 4GL does the following:

1. Restores the screen to its state before the external program began.
2. Resumes the INPUT statement at the BYTE or TEXT field.
3. Reactivates any key sequences specified in the ON KEY clause.

Completing the INPUT ARRAY Statement

The following actions can terminate the INPUT ARRAY statement:

- The user presses one of the following keys:
  - The Accept, Interrupt, or Quit key
  - The RETURN or TAB key from the last field (and INPUT WRAP is not currently set by the OPTIONS statement)
- 4GL executes the EXIT INPUT statement.

By default, the Accept, Interrupt, or Quit keys terminate the INPUT ARRAY statement. Each of these actions also deactivates the form. (But pressing the Interrupt or Quit key can immediately terminate the program, unless the program also includes the DEFER INTERRUPT and DEFER QUIT statements.)

The user must press the Accept key explicitly to complete the INPUT ARRAY statement under the following conditions:

- INPUT WRAP is specified in the OPTIONS statement.
- An AFTER FIELD block for the last field includes a NEXT FIELD clause.

If 4GL previously executed a DEFER INTERRUPT statement in the program, an Interrupt signal causes 4GL to do the following:

- Set the global variable **int_flag** to a nonzero value.
- Terminate the INPUT ARRAY statement, but not the 4GL program.

If 4GL previously executed a DEFER QUIT statement in the program, a Quit signal causes 4GL to do the following:

- Set the global variable **quit_flag** to a nonzero value.
- Terminate the INPUT ARRAY statement, but not the 4GL program.

Executing Control Blocks when INPUT ARRAY Terminates

When INPUT ARRAY terminates, control blocks are executed in this order:

1. The AFTER FIELD clause for the current field.
2. The AFTER ROW clause.
3. The AFTER INPUT clause.

If INPUT ARRAY is terminated by the EXIT INPUT keywords, or by pressing the Interrupt or Quit keys, 4GL does not execute any of these clauses. If a NEXT FIELD statement appears in one of these clauses, 4GL places the cursor in the specified field and returns control to the user.

The INPUT ARRAY statement on the next page supports data entry into a screen form.

The BEFORE FIELD clauses display messages telling the user what to enter in the **stock_num**, **manu_code**, and **quantity** fields. The AFTER FIELD clauses check that user entered values for the **stock_num**, **manu_code**, and **quantity** fields. When the user enters item values for the **stock_num** and **manu_code** fields, 4GL calls **get_item( )** to display a description and price of the item. When all three fields are specified, 4GL displays the total cost.

In this example, the BEFORE INSERT, AFTER INSERT, and AFTER DELETE clauses call functions that ensure that the numbering of the items is accurate. This is necessary because the user can press the Insert and Delete keys at run time to insert and to delete items within the screen form.

```
CALL SET_COUNT(1)
INPUT ARRAY p_items FROM s_items.*
    BEFORE FIELD stock_num
        MESSAGE "Enter a stock number."
    BEFORE FIELD manu_code
        MESSAGE "Enter the code for a manufacturer."
    BEFORE FIELD quantity
        MESSAGE "Enter a quantity"
    AFTER FIELD stock_num, manu_code
        MESSAGE ""
        LET pa_Curr = arr_curr()
        IF p_items[pa_curr].stock_num IS NOT NULL
            AND p_items[pa_curr].manu_code IS NOT NULL THEN
                CALL get_item()
                IF p_items[pa_curr].quantity IS NOT NULL THEN
                    CALL get_total()
                END IF
        END IF
    AFTER FIELD quantity
        MESSAGE ""
        LET pa_curr = arr_curr()
        IF p_items[pa_curr].stock_num IS NOT NULL
            AND p_items[pa_curr].manu_code IS NOT NULL
            AND p_items[pa_curr].quantity IS NOT NULL THEN
                CALL get_total()
        END IF
    BEFORE INSERT
        CALL get_item_num()
    AFTER INSERT
        CALL renum_items()
    AFTER DELETE
        CALL renum_items()
END INPUT
```

References

DEFER, DISPLAY ARRAY, INPUT, OPEN WINDOW, OPTIONS, SCROLL

LABEL

The LABEL statement defines a *statement label*, marking the next statement as one to which the WHENEVER statement or the GOTO statement can transfer program control.

LABEL *label identifier* **:**

label identifier is a statement label. A colon (:) follows the last character.

Usage

The LABEL statement indicates where to transfer control of program execution within the same program block. Upon executing a GOTO or WHENEVER statement that references the *label identifier*, 4GL jumps to the statement immediately following the LABEL statement, skipping any intervening statements. (See also the GOTO statement, on page 3-122, for suggestions regarding the use of GOTO and LABEL statements.)

The following restrictions apply to the LABEL statement:

- The *label identifier* must be unique among labels in its program block.
- To jump to a label, the GOTO or WHENEVER statement must use the same *label identifier* as the LABEL statement above the desired statement.
- The GOTO and LABEL (or the WHENEVER and LABEL) statements must both be in the same MAIN, FUNCTION, or REPORT program block.

The *label identifier* must follow the rules for 4GL identifiers, as described on page 2-9. A colon (:) symbol must immediately follow the last character. You may wish to declare a meaningful name to indicate something about the purpose of the jump:

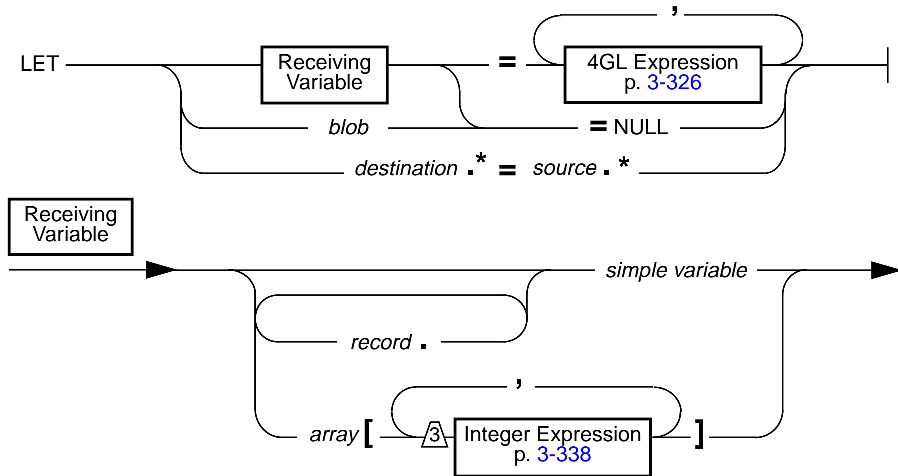
```
WHENEVER ERROR GO TO :l_error
    ...
LABEL l_error:
ERROR "Cannot complete processing."
ROLLBACK WORK
```

References

GOTO, WHENEVER

LET

The LET statement assigns a value to a variable, or a set of values to a record.



array	is the name of a variable of the ARRAY data type.
blob	is the name of a variable of the BYTE or TEXT data type.
destination, source	are names, respectively, of a program record to be assigned values, and of a program record from which to copy values.
record	is the name of a variable of the RECORD data type.
simple variable	is the name of a variable of a simple data type (page 3-295), or a simple member of a record, or element of an array.

Usage

The LET statement can assign a single value to a single variable, or it can assign a set of values from a RECORD variable to an entire program record:

To execute a LET statement, 4GL evaluates the expression on the right of the equal sign (=) and assigns the resulting value to the variable on the left. For example, these statements create the SELECT statement for a PREPARE statement. The backslash is used to embed a quote in a character string:

```
DEFINE sel_stmt CHAR(80)
LET sel_stmt = "SELECT * FROM customer WHERE lname MATCHES \"",
    last_name CLIPPED, "\""
PREPARE s1 FROM sel_stmt
```

This example assigns a NULL value to a variable of the MONEY data type:

```
DEFINE total_price MONEY
LET total_price = NULL
```

You can use most of the 4GL built-in functions and character operators like CLIPPED and USING within the LET statement. For example, these statements use the ASCII operator to ring the terminal bell (= the ASCII value for 7):

```
DEFINE bell CHAR(1)
LET bell = ASCII 7
DISPLAY bell
```

You cannot assign individual values to an entire program record nor to a program array. You cannot use the THRU or THROUGH notation (page 3-363) in the LET statement, but you can assign all the values of one program record to another program record of the same size by using the asterisk (*) notation:

```
LET  x.*  =  y.*
```

This copies the value of each member of the **y** record to consecutive members of the **x** record. The two records must have the same number of members, and corresponding members must be of compatible data types (page 3-324).

To reference substrings of CHAR or VARCHAR variables, specify the starting and ending character positions as integers. These substring positions must be enclosed within brackets and separated by a comma, as in this example:

```
DEFINE full_name CHAR(20), first_name CHAR(10)
LET full_name[1,10] = new_first
```

For TEXT and BYTE variables, LET can assign only NULL values. The LET statement cannot assign any other values to TEXT and BYTE variables. To assign values to these variables, you can do one of the following:

- Use the INTO clause of the SELECT, FOREACH, OPEN, or FETCH statement.
- Pass the name of the variable as an argument to a function.

4GL performs data type conversion on compatible data types (page 3-319).

NLS

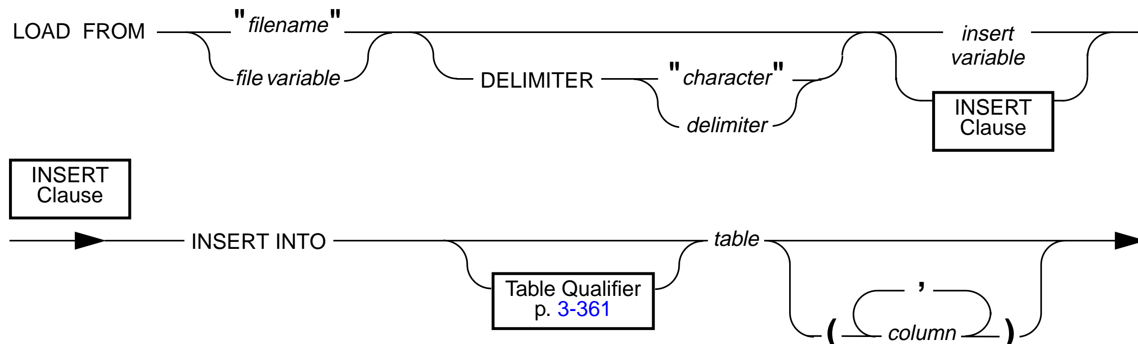
If NLS is active and you convert numeric or monetary values by using the LET statement, the conversion process inserts locale-specific separators and currency symbols into the created strings, not US English separators and currency symbols. Monetary values take on separators and currency symbols specified by LC_MONETARY. Numeric values take on separators specified by LC_NUMERIC. This happens regardless of you including a USING clause in the LET statement. However, if DBFORMAT or DBMONEY is set, these settings override settings in LC_ variables.

References

GLOBALS, INITIALIZE

LOAD

The LOAD statement inserts data from an ASCII file into an existing table.



character is a quoted string, specifying a delimiter symbol.

column is the name of a column in *table*, in parentheses. If you omit the list of column names, the default is all the columns of *table*.

delimiter is a CHAR or VARCHAR variable containing a symbol.

filename is the specification (within quotation marks) of a file that contains the input data. This can include a pathname.

file variable is a CHAR or VARCHAR variable containing a *filename*.

insert variable is a CHAR or VARCHAR variable containing an INSERT clause.

table is the name of a table, synonym, or view in the current database (as specified by a prior DATABASE statement in the same module), or in a database specified in the table qualifier.

Usage

The LOAD statement must include an INSERT statement (either directly or in a variable) to specify where to store the data. LOAD appends the new rows to the specified table, synonym, or view. It does not overwrite existing data. It cannot add a row that has the same key as an existing row. You cannot use the PREPARE statement to preprocess a LOAD statement.

The user who executes the LOAD statement must have Insert privileges for *table*. (For information on database-level and table-level privileges, see the *Informix Guide to SQL: Reference*.)

The Input File

The variable or string following the LOAD FROM keywords must specify the name of a file of ASCII characters that represent the data to be inserted.

How data values in this input file should be represented by a character string depends on the SQL data type of the receiving column in *table*:

Data Type	Input Format
CHAR, VARCHAR, TEXT	Values can have more characters than the declared maximum length of the column, but any extra characters are ignored. The backslash (\ ) symbol is required before any literal backslash symbol or any literal delimiter character, and before any NEWLINE character anywhere in a VARCHAR value, or as the last character in a TEXT value. Blank values can be represented as one or more blank characters between delimiters, but leading blanks must not precede other CHAR, VARCHAR, or TEXT values.
DATE	Values must be in the format *month* **/** *day* **/** *year* (page 3-349) unless some other format is specified by the DBDATE environment variable. You must represent the month as a two-digit number. You can use a two-digit number for the year if the year is in the range 1900 to 1999. Values must be actual dates; for example, February 30 is illegal.
DATETIME, INTERVAL	Values must be in numeric format *year* **-** *month* **-** *day* *hour* **:** *minute* **:** *second* **.** *fraction* (as on page 3-351 and 3-355), without DATETIME or INTERVAL keywords or qualifiers. Time units outside the declared column precision can be omitted. The DATETIME *year* must be a four-digit number; other units (except *fraction*) require two digits.
MONEY	Values can have leading currency symbols, but these are not required.
SERIAL	Values can be represented as `0` to tell the database engine to supply a new SERIAL value. You can specify a literal integer (page 3-340) greater than zero, but if the column has a unique index, an error results if this number duplicates an existing value.
BYTE	Values must be in ASCII-hexadecimal form, without leading or trailing blanks.

Each set of data values in *filename* that represents a new row is called an *input record*. The NEWLINE character must terminate each input record in *filename*. Specify only values that **4GL** can convert to the data type of the database column. Inserted values are truncated from the right if they exceed the declared length of the corresponding CHAR or VARCHAR column.

NULL values of any data type must be represented by consecutive delimiters in the input file; you cannot include anything between the delimiter symbols.

This example shows two records in a hypothetical input file called **nu_cus**:

```
0|Jeffery|Padgett|Wheel Thrills|3450 El Camino|Suite 10|Palo Alto|CA|94306||
0|Linda|Lane|Palo Alto Bicycles|2344 University||Palo Alto|CA|94301|(415)323-6440
```

This **nu_cus** data file illustrates the following features of LOAD:

- The first data value in each record is zero (0), because the engine should here supply a value for a SERIAL column in the row of the database table.
- The vertical (|) bar, the default delimiter, separates consecutive values.
- It uses adjacent delimiters to assign NULL values to the **phone** column in the first record and to the **address2** column for the second record.

The following LOAD statement inserts all the values from the **nu_cus** file into a **customer** table that is owned by the user whose login is **krystl**:

```
LOAD FROM "nu_cus" INSERT INTO krystl.customer
```

Each input record must contain the same number of delimited data values. If you do not include a list of columns in the INSERT clause, the sequence of values in each input record must match the columns of *table* in number and order. Each value must correspond to the literal format of the column data type, or the format of a compatible data type (page 3-324).

A file created by the UNLOAD statement (page 3-274) can be used as input for the LOAD statement, if its values are compatible with the schema of *table*.

The **onload** and **dbload** utilities give you more flexibility for the format of the input file. See the *INFORMIX-Online Dynamic Server Administrator's Guide*, Version 6.0 for a description of **onload** or the *INFORMIX-SE Administrator's Guide*, Version 6.0 for a description of **dbload**.

NLS

The LOAD statement expects incoming data in the format specified by environment variables DBFORMAT, DBMONEY, LC_NUMERIC, LC_MONETARY, and DBDATE. The precedence of these format specifications is consistent with that of other output facilities (that is, forms and reports). If there is an inconsistency, an error is reported and the LOAD is cancelled. For more information, see Appendix E.

The DELIMITER Clause

The DELIMITER clause specifies the ASCII symbol that must separate consecutive data values within each input record. The next statement, for example, identifies the caret (^) symbol as the delimiter:

```
LOAD FROM "/a/data/ord.loadfile" DELIMITER "^"
    INSERT INTO orders
```

If you omit this clause, the character specified by the DBDELIMITER environment variable is used. If the DBDELIMITER variable has not been set, the default delimiter is the vertical bar (| = ASCII 124). See *Informix Guide to SQL: Reference* for information about how to set the DBDELIMITER variable.

You *cannot* specify any of the following characters as the delimiter symbol:

- Hexadecimal numbers (`0` through `9` , `a` through `f`, or `A` through `F`)
- NEWLINE or Control-J
- Backslash (\ ) symbol

The backslash (\ ) symbol serves as an escape character to indicate that the next character is to be interpreted literally as part of the data, rather than as a delimiter or record separator or escape character. If any character value in the input file includes the delimiter or NEWLINE symbols without backslashes, the LOAD statement produces error -846; see *Informix Error Messages, Version 6.0*.

***Note:** The UNLOAD statement automatically inserts a backslash (\ ) before literal delimiter or NEWLINE symbols in character values. When the LOAD statement (or the **onload** or **dbload** utility) inserts output from the UNLOAD statement into a database table, then these escapist backslash symbols are automatically stripped.*

The INSERT Clause

The INSERT clause specifies the table and columns in which to store the new data. This clause supports a subset of the syntax of the INSERT statement, which is described in the *Informix Guide to SQL: Reference.* You cannot, however, include the VALUES, SELECT, nor EXECUTE PROCEDURE clauses of the INSERT statement within the INSERT clause of the LOAD statement. You must specify explicit column names if either of these conditions is true:

- You are not inserting data into all of the columns of *table.*
- The input file does not match the default order of columns, as listed in the **syscolumns** table of the system catalog.

The following example identifies the **price** and **discount** columns as the only columns into which to insert non-NULL data values:

```
LOAD FROM "/tmp/prices" DELIMITER ","
    INSERT INTO maggie.worktab(price,discount)
```

If LOAD is executed within a transaction, the inserted rows are locked, and they remain locked until the COMMIT WORK or ROLLBACK WORK statement terminates the transaction. If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the LOCK TABLE statement after the transaction begins. This exclusive table lock is released when the transaction terminates. (Transactions, row locking, and table locking are described in *Informix Guide to SQL: Tutorial.*)

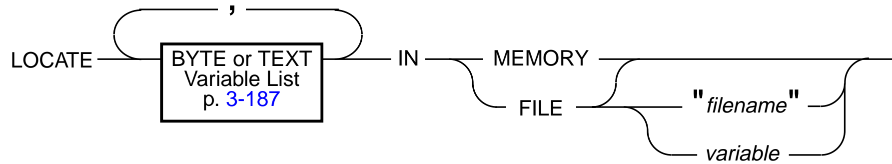
Note: *Consult the documentation of your database engine for details of the limit on the number of locks available during a single transaction.*

References

DATABASE, UNLOAD

LOCATE

The LOCATE statement specifies where to store a TEXT or BYTE value.



filename is the name of a file in which to store the TEXT or BYTE value. This specification can include a pathname and file extension.

variable is the name of a CHAR or VARCHAR variable containing a *filename* specification. (This can also be a CHAR or VARCHAR member of a record, or element of an array, as illustrated on page 3-189.)

Usage

The TEXT or BYTE variable that stores a large binary value is also called a blob (for *binary large object*). You must specify whether you want to store the value of the variable in memory or in a file. You can access a value from memory faster than from a file. If your program exceeds the available memory, however, 4GL automatically stores part of the blob value in a file. To use a blob variable, your program must do the following:

1. Declare the variable with a DEFINE statement.
2. Use the LOCATE statement to specify the storage location. The LOCATE statement must appear within the scope of reference of the variable.

The following topics are described in this section:

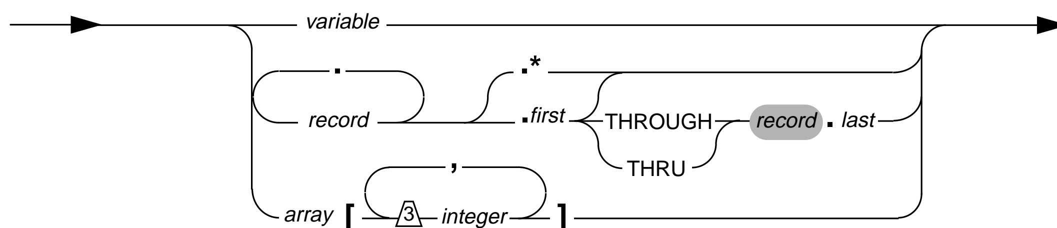
Topic	Page
The List of Large Variables	3-187
The IN MEMORY Option	3-187
The IN FILE Option	3-188
Using a Temporary File	3-188
Specifying a Filename	3-189
Passing Large Variables to Functions	3-189
Freeing the Storage Allocated to a Large Variable	3-190

The LOCATE statement must follow any DEFINE statement that declares TEXT or BYTE variables, and it must appear in the same program block as a local TEXT or BYTE variable. If you try to access a TEXT or BYTE value before initializing its variable with a LOCATE statement, 4GL generates a run-time error.

The List of Large Variables

This comma-separated list specifies the large variable(s) to be initialized:

BYTE or TEXT Variable List



array is the name of a structured variable of the ARRAY data type.

first is the name of a large member variable to be initialized.

integer is a literal integer between 0 and the declared size of the array.

last is another member of *record* that was declared later than *first.*

record is the name of a structured variable of the RECORD data type.

variable is the name of a large variable of the TEXT or BYTE data type that was declared in a previous DEFINE statement.

You can then use most 4GL statements to access the variable. The LET statement can assign a NULL value to a TEXT or BYTE variable, but it cannot assign non-NULL values. The INTO clause of the SELECT statement can assign to a specified variable a TEXT or BYTE value from the database.

The IN MEMORY Option

Use the IN MEMORY option to allocate storage in memory for TEXT or BYTE values. The following example declares the variable **quarter** as the same data type as the database column **analysis**, and stores the variable in memory:

```
DEFINE quarter LIKE stock.analysis
LOCATE quarter IN MEMORY
```

If the TEXT or BYTE variable has already been stored in memory, then using the LOCATE statement again reinitializes the variable.

If a TEXT or BYTE variable has been initialized to memory or to a temporary file, you can use LOCATE to reinitialize the variable. You *cannot* reinitialize a TEXT or BYTE variable that is stored in a named file.

The IN FILE Option

The IN FILE option stores the TEXT or BYTE values in a file. 4GL opens and closes the file each time that you use the variable in an SQL or other 4GL statement. When you retrieve a row containing a TEXT or BYTE column, the value from the database column overwrites the current contents of the file. Similarly, when you update a row, 4GL reads and stores the entire contents of the file in the database column. As with storage in memory, the file contains only the value most recently assigned to the variable. You have several options with the IN FILE clause:

- Omit a filename, so that 4GL places the value in a temporary file.
- Specify a variable that contains the name of a file in which to store the TEXT or BYTE value. The *filename* can include a pathname.

These options are described in the sections that follow.

Using a Temporary File

If you omit the filename, 4GL places the TEXT or BYTE value in a temporary file. 4GL creates the temporary file at run time in the directory identified by the DBTEMP environment variable. If DBTEMP is not set, 4GL puts the temporary files in the **/tmp** directory. If no temporary directory exists, a run-time error occurs.

The following example omits the filename. It also shows that TEXT and BYTE types can be declared as components of RECORD variables:

```
DEFINE stock RECORD
        n INTEGER, analysis TEXT, graph BYTE
    END RECORD
LOCATE stock.analysis IN FILE
LOCATE stock.graph IN FILE
```

If the TEXT or BYTE variable has already been located in a temporary file, then using the LOCATE statement again reinitializes the variable.

You can specify *multiple filenames* by declaring an array of character variables. This example stores an array of filenames in an array of TEXT variables:

```
DEFINE flnames ARRAY[10] OF char(20),
    t_holds ARRAY[10] of TEXT
    i INTEGER
FOR i = 1 TO 5
    LET flnames[i] = "/u/db/profile", i, USING "<<&"
    LOCATE t_holds[i] IN FILE flnames[i]
END FOR
```

Specifying a Filename

To place the TEXT or BYTE value in a specific file, the LOCATE statement can include either a literal filename, or else a character variable that contains the filename. This example uses a quoted string to specify the filename:

```
DEFINE analysis TEXT
LOCATE analysis IN FILE "/u/db/analysis"
```

The next example uses a CHAR variable to specify the filename:

```
DEFINE flname CHAR(20),t_hold TEXT
LET flname = "/tmp/TodaysReport"
LOCATE t_hold IN FILE flname
```

Passing Large Variables to Functions

If you specify a variable a variable in the argument list of a function or report, 4GL ordinarily passes it by value. The function or report can modify the passed value without affecting the variable in the calling function.

4GL handles large binary objects ("blobs," which are variables of the TEXT and BYTE data types) differently. It passes blob variables *by reference*. This means that if a function modifies a TEXT or BYTE variable, the change is apparent to the variable in the calling routine. The CALL statement (page 3-16) need not include a RETURNING clause for a TEXT or BYTE value.

Freeing the Storage Allocated to a Large Variable

If you no longer need a TEXT or BYTE variable, you can use the following statements to release the memory that stored its value:

Statement	Description
FREE	If you stored the TEXT or BYTE variable in a file, you can reference the variable in the FREE statement to delete the file. If you stored the TEXT or BYTE variable in memory, the FREE statement releases all memory associated with the variable.
LOCATE	The LOCATE statement for the same variable releases memory and removes temporary files, but does not remove named files.

When it encounters the RETURN statement or the END FUNCTION, or END REPORT keywords, 4GL frees any *local* TEXT or BYTE variables that are stored in memory or in a temporary file. 4GL does not, however, de-allocate storage for TEXT or BYTE variables that are passed by reference as arguments to a function or to a report. Storage for such variables is de-allocated when EXIT PROGRAM or END MAIN terminates the program. 4GL does not automatically remove a named file that is associated with a TEXT or BYTE variable.

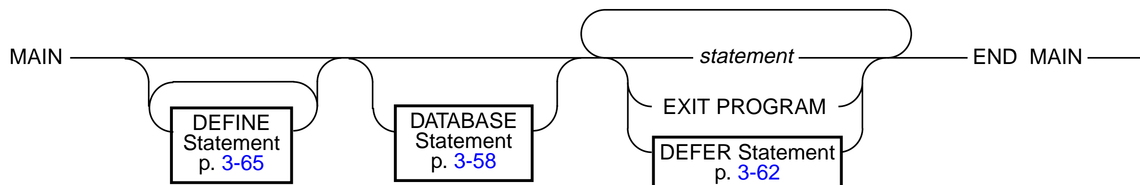
After you release the storage, you cannot access the TEXT or BYTE variable without executing a new LOCATE statement to initialize it. If you have named the file for the TEXT or BYTE value, and you want to retain the file, you should not use the FREE statement. For information on the FREE statement, see the *Informix Guide to SQL: Reference*.

References

DEFINE, EXIT, FUNCTION, GLOBALS, MAIN, INITIALIZE, REPORT, RETURN

MAIN

The MAIN statement defines the MAIN program block.



statement is any SQL statement or other 4GL statement (but not FUNCTION, GLOBALS, MAIN, REPORT, RETURN, nor the 4GL report execution statements NEED, PAUSE, PRINT, and SKIP).

Usage

Every 4GL program must have exactly one MAIN statement. The MAIN statement typically calls functions or reports to do the work of the application. The following program fragment illustrates a program that calls functions whose definitions appear in the same module as the MAIN statement:

```
MAIN
    ...
    CALL get_states()
    CALL ring_menu()
    ...
END MAIN

FUNCTION get_states()
    ...
END FUNCTION

FUNCTION ring_menu()
    ...
END FUNCTION
```

The MAIN statement cannot appear within another statement. It must be the first program block of the module in which it appears, as in this example.

The END MAIN Keywords

The END MAIN keywords mark the end of the MAIN program block. The program terminates when it encounters these keywords. If it encounters the EXIT PROGRAM statement, however, the program terminates before END MAIN.

Variables Declared in the MAIN Statement

You can declare variables by including DEFINE statements within the MAIN program block. Variables that you declare here are *local* to the MAIN block; you cannot reference their names in any FUNCTION or REPORT definition.

If you include a DEFINE statement before the MAIN statement, however, and outside of any FUNCTION or REPORT statement, its *module* variables are visible to subsequent statements in any other program block of the same source module. The GLOBALS statement (page 3-117) can extend the visibility of a module variable beyond the module where it is declared.

If you assign the same identifier to variables that differ in scope of reference, then in any portion of your program where the scopes of their names overlap, the following rules of precedence apply:

- A local variable has the highest precedence, so that within its scope, no identical identifier of a global or module variable can be visible.
- Within the module in which it was declared, a module identifier takes precedence over another with the same identifier whose scope has been extended by the GLOBALS *filename* statement.

You should assign unique names to global and module variables that you intend to reference within the MAIN program block.

DEFER and DATABASE Statements and the MAIN Program Block

DEFER statements (page 3-62) can appear only within the MAIN statement.

Any DATABASE statement that appears before the MAIN statement (but in the same module) specifies the *default database* at compile time (page 3-59). The same database also becomes the *current database* at run time (page 3-60), unless another DATABASE statement specifies a different database.

Any DATABASE statement within the MAIN statement must follow the last DEFINE declaration (page 3-65). The specified database becomes the current database for subsequent SQL statements until the program ends, or until another DATABASE statement is encountered.

References

CALL, DATABASE, DEFER, DEFINE, EXIT PROGRAM, FUNCTION, GLOBALS, REPORT

MENU

The MENU statement creates and displays a ring menu. By using the keyboard, the user can choose menu options that execute blocks of statements.

MENU "*title*" / *variable* MENU Control Block p. 3-194 END MENU

title a quoted string that specifies a display label for the ring menu.

variable is a CHAR or VARCHAR variable containing a display label.

Usage

This statement specifies and displays a ring menu that occupies two lines in the current 4GL window. The MENU statement can do the following:

- Create and display a screen menu, including a *title* of the menu.
- Define and display a list of menu *options* that the user can select.
- Specify a single-line *description* to display for each menu option.
- Designate *Help message* numbers for each menu option.
- Specify a block of 4GL *statements* to execute for each menu option.

When it encounters a MENU statement, 4GL performs these actions:

1. Displays in the current 4GL window the title and the options of the menu.
2. Moves the *menu cursor* to the first option, and displays its description. (The order of options is determined by their order in the control blocks. The menu cursor marks the current menu option with a double border.)
3. Waits for the user to press the activation key for a MENU control block, or to terminate the MENU statement by pressing the Quit or Interrupt key.
4. If the user presses an activation key, 4GL executes the statements in the corresponding control block, until it encounters one of these statements:
   - EXIT MENU statement. 4GL then exits from the menu.
   - CONTINUE MENU statement. 4GL skips any remaining statements in the MENU control block, and redisplays the menu.
   - Last statement in the MENU control block. 4GL redisplays the menu so that the user can choose another option.

A menu can appear above or below a screen form, but not within a form. 4GL displays the menu title and the menu options on the Menu line. This

reserved line is positioned by the most recent MENU LINE specification in the OPTIONS or OPEN WINDOW statement. The default position is to the *first* line of the current 4GL window.

Unless the title and at least one option can fit on the screen or in the current 4GL window, a run-time error occurs. For information on multiple-page menus, and how the set of menu options act like a "ring" for the menu cursor, see "Scrolling the Menu Options" on page 3-208.

The *title* of a menu is just a display label; you cannot reference a menu by name. To repeat the same menu and all its behavior in different parts of a program, you can include the MENU statement in a FUNCTION definition, so that you can invoke the function when you want the menu to appear.
The following topics are described in this section:

Topic	Page
The MENU Control Blocks	3-194
The BEFORE MENU Block	3-196
The COMMAND Clause	3-197
The HELP Clause	3-197
The KEY Clause	3-198
Invisible Menu Options	3-200
The CONTINUE MENU Statement	3-201
The EXIT MENU Statement	3-202
The NEXT OPTION Clause	3-203
The HIDE OPTION and SHOW OPTION Keywords	3-203
The END MENU Keywords	3-205
Identifiers in the MENU Statement	3-205
Choosing a Menu Option	3-206
Scrolling the Menu Options	3-208
Completing the MENU Statement	3-209

The MENU Control Blocks

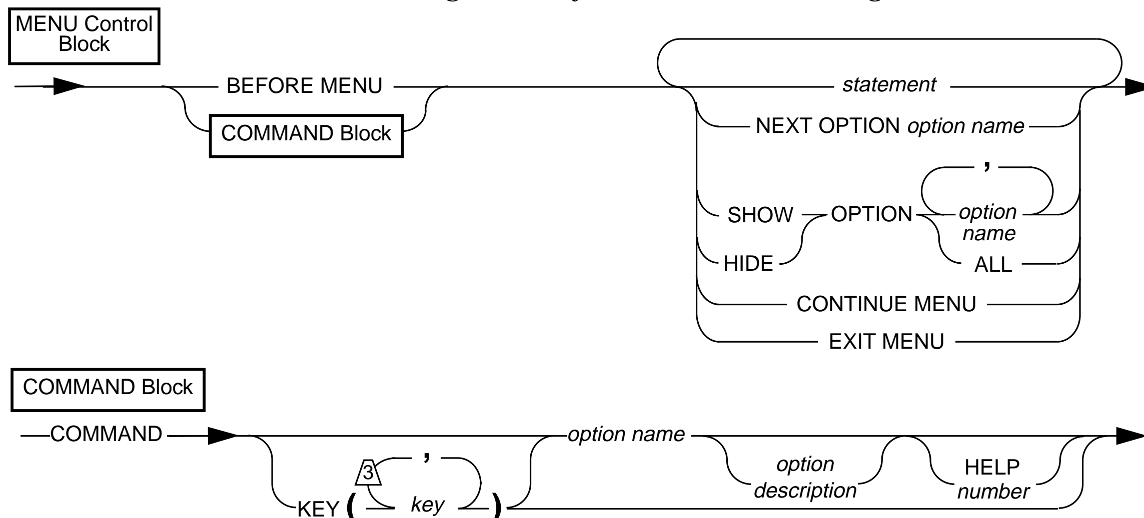
Each control block includes a *statement block* of at least one statement, and an *activation clause* that specifies when to execute the statement block. Any of three types of activation clauses can appear within MENU control blocks:

- BEFORE MENU clause (to execute the block *before* the menu is displayed).
- COMMAND *option* clause (to specify the *name* and *description* of an *option*, an optional *activation key*(*s*) to choose the option, and an optional *Help message* code; execute the block when the user chooses the option).
- *Hidden option* clause (a COMMAND clause that only specifies *activation key*(*s*) to execute a statement block if the *key* is pressed; no option name, option description, nor Help message number is specified).

The statement block can specify SQL or other 4GL statements to execute when a user presses a key sequence, as well as special MENU instructions:

- The next menu option to highlight with the menu cursor.
- Whether to suppress or restore the display of one or more menu options.
- Whether to exit from the MENU statement.

The activation clause and statement block correspond respectively to the left-hand and right-hand syntax elements in the diagram that follows:



key is a letter, a literal symbol in quotation marks, or a keyword. (Quotation marks are not required if *key* is a single letter of the alphabet.) This list of up to four (4) activation keys must be enclosed in parentheses; see "The KEY Clause" on page 3-198.

number is an integer that identifies the Help message for this menu option. You must have used the OPTIONS statement previously to identify the Help file containing the message.

option description is a quoted string or the name of a CHAR or VARCHAR variable that contains an *option description* for the Menu help line.

option name is a quoted string or the name of a CHAR or VARCHAR variable that contains the *name* of the menu option. This cannot be longer than the width of the current 4GL window.

statement is an SQL statement or other 4GL statement.

The screen displays a ring menu of *option names* as menu options. The menu options appear in the order in which you specify them in COMMAND clauses of the MENU statement. You must include at least one option (that is, one

COMMAND clause) for each menu. Within the MENU control block that includes the COMMAND clause, you can include statements that perform the activity that is described by the menu option and its description.

The *option description* appears on the line below the menu when the option is current. The string length must not be longer than the width of the screen or 4GL window. See also "Identifiers in the MENU Statement" on page 3-205.

The BEFORE MENU Block

Before displaying the menu, 4GL executes any statements in the statement block that follows the optional BEFORE MENU keywords. Use statements in this control block to perform preliminary tasks, such as the following:

- Specifying values for variables used for the menu name, the names of options, and the strings containing descriptions of options.
- Hiding or showing individual menu options.
- Checking user access privileges.

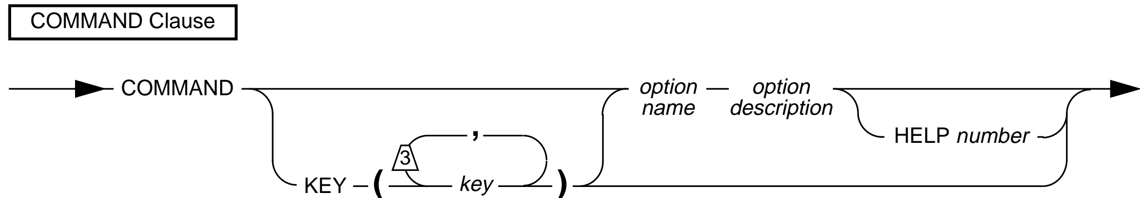
If 4GL encounters the EXIT MENU statement here, no menu is displayed.

The following program fragment includes statements that specify the name of the menu, the name of an option, and the option description at run time:

```
DEFINE menu_name, opt_name CHAR(20)
    opt_desc CHAR(40), priv_flag SMALLINT
LET menu_name = "SEARCH"
LET opt_name = "Query"
LET opt_desc = "Query for customers."
IF ...
    LET priv_flag = 1
END IF
MENU menu-name
    BEFORE MENU
        IF priv_flag THEN
            LET menu_name = "POWER SEARCH"
            LET opt_name = "Power Query"
        END IF
    COMMAND opt_name opt_desc HELP 12
        IF priv_flag THEN
            CALL cust_find(1)
        ELSE
            CALL cust_find(2)
        END IF
    ...
END MENU
```

The COMMAND Clause

The COMMAND clause can define a menu *option* that appear after the menu title in the Menu line, as well as its description that appear in the following line when the menu cursor is on the option. It has the following syntax:



For definitions of these terms, see "The MENU Control Blocks" on page 3-194.

Each COMMAND clause is part of a MENU control block whose statements perform the activity described by the option description. If you want to nest menus, you can include another MENU statement. The MENU control blocks may be easier to read if you use function calls to group statements.

The COMMAND clause can optionally include a HELP clause to associate a Help message number with the menu option. It can also include a KEY clause, to specify up to four *activation key*(s) that the user can press to choose the option; otherwise, 4GL recognizes *default activation keys*, as the next page describes.

The HELP Clause

The HELP clause specifies the *number* of a Help message to display for the menu option. 4GL displays the Help message when the menu option is highlighted and the user presses the Help key. The default Help key is CONTROL-W. You can use the OPTIONS statement to assign a different Help key.

The following MENU statement specifies different Help message numbers for each of two menu options:

```
MENU "MAIN"
    COMMAND "Customer" "Enter and maintain customer data"
            HELP 101
        CALL customer( )
        CALL ring_menu( )
    COMMAND "Orders" "Enter and maintain orders" HELP 102
        CALL orders( )
        CALL ring_menu( )
    ...
END MENU
```

You can create Help messages (and their numbers) in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement. Use the **mkmessage** utility to create a run-time version of the Help file. A run-time error occurs if the Help file cannot be opened, or if you specify a Help number that does not occur in the Help file, or that is greater than 32,767.

The KEY Clause

The KEY clause in a MENU control block specifies *activation keys* that users can press to choose the option (if an *option name* is specified) and to execute the statements in the MENU control block. If you omit the KEY clause, the first character in *option name* is the default activation key to choose the option.

If a user chooses the option, 4GL executes the statements in the MENU control block that includes the COMMAND clause. If EXIT MENU is not among these statements, 4GL redisplays the menu, so the user can choose another option.

This MENU statement, for example, creates a menu entitled **TOP LEVEL** that displays five options. The default activation keys are `a`, `f`, `c`, `d`, and `e`:

```
MENU "TOP LEVEL"
    COMMAND "Add" "Add a row to the database."
        ...
    COMMAND "Find" "Find a row in the database."
        ...
    COMMAND "Change" "Update a row in the database."
        ...
    COMMAND "Delete" "Delete a row from the database."
        ...
    COMMAND "Exit" "Return to the operating system."
        EXIT MENU
END MENU
```

This MENU statement produces the following initial display:

```
TOP LEVEL: [Add] Find  Change  Delete  Exit
Add a row to the database
```

One option is always marked as the *current option*. This option is marked by a double border, called the *menu cursor*.

The line under the menu options (the *Menu help line*) displays a description of the menu option, as specified in the COMMAND clause for that menu option. If the menu cursor moves to another option, the display in this line changes, unless you specify the same description for both menu options.

4GL executes the statements in the MENU control block if the user presses an *activation key* that you specify by *key* specification in the KEY clause:

- Letters. (Both upper and lowercase letters are valid, but 4GL does not distinguish between them.)
- Symbols (such as !, @, or #) enclosed between quotation marks.
- Any of the following keywords (in uppercase or lowercase characters):

DOWN	INTERRUPT	RETURN *or* ENTER	TAB
ESC *or* ESCAPE	LEFT	RIGHT	UP
F1 *through* F64			
CONTROL-*char* (except A, D, H, I, J, L, M, R, or X)			

The following keys deserve special consideration before you assign them as activation keys in the KEY clause of a MENU statement:

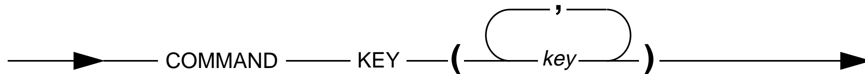
Key	**Special Considerations**
ESC or ESCAPE	You must use the OPTIONS statement to specify another key as the Accept key, because this is the default Accept key.
INTERRUPT	You must include a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, **4GL** executes the statements in the MENU control block and sets **int_flag** to nonzero, but does not terminate the MENU statement. 4GL also executes the statements in the control block if DEFER QUIT has been executed and the user presses the Quit key. In this case, 4GL sets **quit_flag** to non-zero.

CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these keys for field editing.
I, J, and M	The usual meanings of these keys (TAB, LINEFEED, and RETURN, respectively) are lost to the user. Instead, the key is trapped by **4GL** and used to trigger the menu option. For example, if CONTROL-M appears in the KEY clause, the user cannot press RETURN to advance the cursor to the next field.

The *key* must be unique among all the KEY clauses of the MENU statement. You may not be able to specify other keys that have special meaning to your operating system.

Invisible Menu Options

You can add an *invisible* option (an option that is never displayed) to a menu by including a KEY clause in the COMMAND clause of the MENU control block, but not specifying an *option name* nor an *option description*.

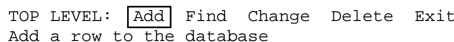


The *key* cannot be the activation key of any other COMMAND clause. If you specify a letter here as the activation key, this must be different from the first character of any option of the same menu.

The following MENU statement creates a menu named **TOP LEVEL** with six options, of which only five appear in the menu display. The exclamation (!) key chooses an invisible option that is not displayed on the menu. Here a description and a Help number are associated with each visible option:

```
MENU "TOP LEVEL"
    COMMAND "Add" "Add a row to the database" HELP 12
        ...
    COMMAND "Find" "Find a row in the database" HELP 13
        ...
    COMMAND "Change" "Update a row in the database" HELP 14
        ...
    COMMAND "Delete" "Delete a row from the database" HELP 15
        ...
    COMMAND KEY ("!")
        CALL bang()
        ...
    COMMAND "Exit" "Return to operating system" HELP 16
        EXIT PROGRAM
END MENU
```

These statements produce the following menu display:



The CONTINUE MENU Statement

The CONTINUE MENU statement causes 4GL to ignore the remaining statements in the current MENU control block, and redisplay the menu. The user can then choose another menu option, as in the following program fragment.

In the next example, the **Yearly Report** option first cautions the user that a report takes several hours to create. If the user types Y to create the report, then 4GL calls the **calc_yearly( )** function. Otherwise, 4GL executes the CONTINUE MENU statement and redisplays the **YEAR END** menu:

```
MENU "YEAR END"
    COMMAND "Yearly Report" "Compile Yearly Statistics Report"
        PROMPT "This report takes several hours to create." ,
            "Do you want to continue? (y/n)" FOR answer
        IF answer MATCHES "[Yy]" THEN
            CALL calc_yearly()
        ELSE
            CONTINUE MENU
    ...
END MENU
```

The EXIT MENU Statement

The EXIT MENU statement terminates the MENU statement without executing any remaining statements in the menu control blocks. Use this statement at any point where you want the user to leave the menu instead of redisplaying it. You must specify this statement for at least one menu option in each 4GL menu. Otherwise, the user will have no way to leave the menu. If it encounters the EXIT MENU statement, 4GL takes the following actions:

- Skips all statements between the EXIT MENU and END MENU keywords.
- Deactivates and erases the menu.
- Resumes execution at the first statement after the END MENU keywords.

The following example demonstrates using the EXIT MENU keywords in the MENU block of a menu option named **Exit**:

```
MENU "CUSTOMER"
    ...
    COMMAND "Exit" "leave the CUSTOMER menu." HELP 5
        EXIT MENU
END MENU
```

(To exit from the current MENU control block without exiting from the MENU statement, use the CONTINUE MENU keywords, rather than EXIT MENU.)

The NEXT OPTION Clause

When 4GL finishes executing the statements in a control block that includes a COMMAND clause, the option just executed remains as the current option. If you want a different option to be the current option, use the NEXT OPTION keywords. The NEXT OPTION clause identifies the name of a menu option to make current. This clause does *not* choose the next menu option; rather, it identifies the next menu option that will be highlighted as the current option. The user can simply press RETURN to choose the current option.

In the following MENU statement, if the user selects the **Query** option, 4GL calls the function **query_data( )**, and redisplays the menu with **Modify** as the current option. To choose the **Modify** option, the user presses RETURN.

```
MENU "CUSTOMER"
    COMMAND "Query" "Search for a customer"
        CALL query_data( )
        NEXT OPTION "Modify"
    ...
    COMMAND "Modify" "Modify a customer"
        ...
END MENU
```

If you want the cursor to move among menu options in a certain order, list their defining COMMAND clauses in the desired order. Use the NEXT OPTION keywords only if you want to deviate from the default left-to-right order of the ring menu. 4GL does not execute any of the statements that follow a NEXT OPTION clause within a MENU control block.

The HIDE OPTION and SHOW OPTION Keywords

Use the HIDE OPTION keywords to conceal some menu options from users. 4GL does not display a *hidden option* in the menu, and does not recognize as valid any keystroke that would otherwise select the option (if it were visible). Such options remain hidden and disabled, until 4GL executes a SHOW OPTION clause that references their *command value*.

The following MENU statement creates a menu with seven options. The **Long_menu** option shows all options; the **Short_menu** options shows only the **Query**, **Long_menu**, and **Exit** options:

```
MENU "Order Management"
    COMMAND "Query" "Search for orders"
        CALL get_orders( )
    COMMAND "Add" "Add a new order"
        CALL add_order( )
    COMMAND "Update" "Update the current order"
        CALL upd_order( )
    COMMAND "Delete" "Delete the current order"
        CALL del_order( )
    COMMAND "Long_menu" "Display all menu options"
        SHOW OPTION ALL
    COMMAND "Short_menu" "Display a short menu"
        HIDE OPTION ALL
        SHOW OPTION "Query", "Long_menu", "Exit"
    COMMAND "Exit" "Exit from the Order Management Form"
        EXIT MENU
END MENU
```

If you specify the options to hide by listing them in character variables, you must assign values to the variables before you can include the variables in a HIDE OPTION clause. (See page 3-205 for more information about variables.)

The ALL keyword in a SHOW OPTION or HIDE OPTION clause specifies all of the menu options that you created in any COMMAND clause.

Use the SHOW OPTION keywords to restore a list of menu options that the HIDE OPTION keywords disabled. By default, 4GL displays all menu options. You only need to use this statement if you have previously specified the HIDE OPTIONS keywords to disable at least one menu option.

4GL displays menu options in the same order that their COMMAND clauses defined them. The order in which a SHOW OPTION clause lists options has no effect on the order of their subsequent appearance in the menu. (3-212 at the end of this section illustrates the SHOW OPTION and HIDE OPTION clauses of MENU control blocks.)

Note: *Do not confuse "hidden options" with "invisible options." Neither appears on the menu, but hidden options cannot be accessed by the user until after they have been enabled by the SHOW OPTION keywords. Invisible options have an activation key, but no command name. Their statement blocks can be accessed by pressing an activation key, but they do not appear in the menu. The HIDE OPTION and SHOW OPTION keywords cannot affect invisible options, since (as their name suggests),*

invisible options are never visible in the menu. You must use some other approach to enable and disable invisible options; for example, you might specify their actions within a conditional statement.

Nested MENU Statements

You can nest MENU statements within MENU control blocks, so that the menus form a “tree” hierarchy. Nested MENU statements can appear either directly in a statement block, or in 4GL functions that are called directly or indirectly when the user chooses options of the enclosing menu.

The END MENU Keywords

Use the END MENU keywords to indicate the end of the MENU statement. The END MENU keywords must follow the last statement in the last MENU control block. These keywords are required in every MENU statement. If you are nesting menus within menus, you must include a separate set of END MENU keywords to mark the end of each MENU statement construct.

If 4GL encounters the EXIT MENU statement within any MENU control block, control of execution is immediately transferred to the first statement that follows the END MENU keywords. (To terminate the current MENU control block without exiting from the MENU statement, use the CONTINUE MENU keywords, rather than END MENU or EXIT MENU.)

Identifiers in the MENU Statement

You can specify a character variable for the following:

- The menu title
- The option name
- The option description
- The NEXT OPTION option name
- The SHOW OPTION or HIDE OPTION option name

Assignment statements can appear before 4GL executes the MENU statement or within the MENU statement. You can specify variable values in the BEFORE MENU block and in one or more of the subsequent MENU control blocks. Make sure, however, that a variable has a value before you include it in the MENU statement.

Keep the following in mind if you change the value of a variable that was used as the menu title or as an option name in a MENU statement:

- 4GL determines the length of the menu title and of each option name when it first displays the menu. This length does not change during the MENU statement. If you subsequently assign a new value to a variable, 4GL displays as much of the new value as can fit in the existing space. For example, suppose that you assign the string **Short_Menu** (10 characters) to a variable, and later specify that variable as a menu title.
  If a subsequent statement in a control block of the same MENU statement assigns the new value **Very_Long_Menu** (14 characters) to the variable, 4GL displays only the first ten characters of the new title.

  Similarly, if a second MENU control block assigns the value `Menu` (four characters) to the variable that you specified as the menu title, 4GL displays the new title with six trailing blank spaces. For examples of using a variable as a menu title, an option name, and an option description in the MENU statement, see the program fragment in the section "Completing the MENU Statement" on page 3-209.)
- If you use an *array element* (for example, **p_array[i]**) as a variable in a MENU statement, be aware that 4GL calculates the value of the index variable only once, *before* it first displays the menu. To index into the array, 4GL uses the value of the index variable after executing the BEFORE MENU block (if that block is included). Any subsequent changes to the index variable made in subsequent MENU control blocks do *not* affect the way that 4GL evaluates the array element variable.

4GL produces a run-time error if the length of a variable or quoted string that specifies a menu name, an option name, or an option description exceeds the width of the current 4GL window.

Choosing a Menu Option

The user can choose a menu option in any of the following ways:

- Using the Arrow keys to position the menu cursor on the option and pressing RETURN. (See also the section "Scrolling the Menu Options" on page 3-208.)
- Typing a key sequence that the KEY clause associated with the option.
- Typing the first letter or letters of the option name (regardless of whether the option is currently displayed on the screen).

When the user types a letter, 4GL looks for a unique match among options:

- If only one option begins with the letter, or only one option is associated in a KEY clause with the letter, the choice is unambiguous. 4GL executes the commands associated with the option.
- If more than one option begins with the same letter, **4GL** clears the second line of the menu and prompts the user to clarify the choice. **4GL** displays each keystroke, followed by the names of the menu options that begin with the typed letters. When **4GL** identifies a unique option, it closes this prompt line and executes the statements associated with the selected menu option.

For example, the next menu includes three options that begin with the letters `Ma`. The following screen is displayed when the user types the letter `M`:

```
Resorts:  Oxnard Malaysia Malta Manteca Pittsburgh Portugal Exit
Select: M ▯ Malay Malta Manteca
```

When the user types the letters `Mal`, 4GL drops **Manteca** from the list and displays the two remaining options:

```
Resorts:  Oxnard Malaysia Malta Manteca Pittsburgh Portugal Exit
Select:   Mal▯Malay Malta
```

At this point, the user can type an `a` to select **Malay** or a `t` to select **Malta**.

The Arrow keys have no effect when choosing among menu options that begin with the same letters. BACKSPACE deletes the keystroke to the left of the cursor.

Scrolling the Menu Options

When 4GL displays a menu, it adds a colon (:) and a blank space after the menu name, and a space before and after each menu option. If the width of the menu exceeds the number of characters that the screen or a 4GL window can display on a single line, 4GL displays the first "page" of options followed by an ellipsis (**...**) symbol. This indicate that additional options exist.For example, the following menu displays an ellipsis (**...**) symbol:

```
menu-name:  menu-option1 menu-option2 menu-option3 menu-option4 ...
optional Help line
```

If the user presses SPACEBAR or [→] to move past the rightmost option (**menu-option4** in this case), 4GL displays the next page of menu options.

In the following example, the ellipses at both ends of the menu indicate that more menu options exist in both directions:

```
menu-name:  ... menu-option5 menu-option6 menu-option7 menu-option8 ...
optional Help line
```

If the user moves the highlight to the right past **menu-option8** in this example, 4GL displays a page of menu options like the following:

```
menu-name:  ... [menu-option9] menu-option6 menu-option10 menu-option11 ...
optional Help line
```

The following keys can move through a menu:

Key	Effect
[→], SPACEBAR	Moves the menu cursor to the *next option*. If the menu displays an ellipsis (. . .) on the right, pressing [→] from the right-most option displays the next page of menu options. If the last menu option is current and no ellipsis is on the right, [→] returns to the first option in the first page of menu options.
[←]	Moves the cursor to the *previous option*. If the menu displays an ellipsis (. . .) on the left, pressing [←] from the leftmost option displays the previous page of menu options. If the first menu option is current and no ellipsis is on the left, pressing [←] returns to the last option on the last page of menu options.
[↑]	Moves the cursor to the first option on the *previous* menu page.
[↓]	Moves the cursor to the first option on the *next* page of menu options.

During interactive statements like INPUT, CONSTRUCT, or INPUT ARRAY, errors would be likely to result if the user could interrupt the interaction with menu choices. **4GL** prevents this by disabling the entire menu during the execution of these statements. The menu does not change its appearance when it is disabled.

Completing the MENU Statement

Any of the following actions can terminate the MENU statement:

- The user chooses the Interrupt key.
- 4GL encounters the EXIT MENU statement.

By default, the Interrupt key terminates program execution immediately. Unlike CONSTRUCT, DISPLAY ARRAY, and INPUT statements, the MENU statement is not terminated by the Interrupt key if 4GL has executed the DEFER INTERRUPT statement. In these cases, an Interrupt signal causes 4GL to do the following:

- Set the global variable **int_flag** to a non-zero value.
- Remain in the MENU statement until EXIT MENU is encountered.

The EXIT MENU statement is typically included in a MENU control block that is activated when the user chooses an **Exit** or **Quit** option, as in the next example. If menus are *nested*, EXIT MENU terminates only the current MENU statement, passing control to the innermost enclosing MENU statement.

In the following program fragment, the MENU statement uses variables for the menu name, for a command name, and for an option description:

```
DEFINE menu_name, command_name CHAR(10),
        option_desc CHAR(30),
        priv_flag SMALLINT

LET menu_name = "NOVICE"
LET command_name = "Expert"
LET option_desc = "Display all menu options."

IF ... THEN
    LET priv_flag = 1
END IF

MENU menu_name
    BEFORE MENU
        HIDE OPTION ALL
        IF priv_flag THEN            -- expert user
            LET menu_name = "EXPERT"
            LET command_name = "Novice"
            LET option_desc = "Display a short menu."
            SHOW OPTION ALL
        ELSE                         -- novice user
            SHOW OPTION "Query", "Detail", "Exit", command_name
        END IF

    COMMAND "Query" "Search for rows." HELP 100
        CALL get_cust()
    COMMAND "Add" "Add a new row." HELP 101
        CALL add_cust()
    COMMAND "Update" "Update the current row." HELP 102
        CALL upd_cust()
        NEXT OPTION "Query"
```

```
    COMMAND "Delete" "Delete the current row." HELP 103
        CALL del_cust()
        NEXT OPTION "Query"
    COMMAND "Detail" "Get details." HELP 104
        CALL det_ord()
        NEXT OPTION "Query"
    COMMAND command_name option_desc HELP 105
        IF priv_flag THEN            -- EXPERT menu visible
            LET menu_name = "NOVICE"
            LET command_name = "Expert"
            LET option_desc = "Display all menu options."
            HIDE OPTION ALL
            SHOW OPTION "Query", "Detail", "Exit", command_name
            LET priv_flag = 0
        ELSE                         -- NOVICE menu visible
            LET menu_name = "EXPERT"
            LET command_name = "Novice"
            LET option_desc = "Display a short menu."
            SHOW OPTION ALL
            LET priv_flag = 1
        END IF
    COMMAND KEY ("!")
        CALL bang()
    COMMAND "Exit" "Leave the program." HELP 106
        EXIT MENU
END MENU
```

These statements produce two menus. This is the “expert” menu:

```
EXPERT:  Query Add Update Delete Detail Novice Exit
Search for rows.
```

This is the simpler "novice" menu:

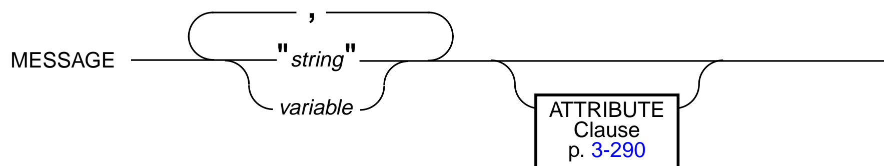
```
EXPERT:  Query Detail Expert Exit
Search for rows.
```

References

CONTINUE, DEFER, OPEN WINDOW, OPTIONS

MESSAGE

The MESSAGE statement displays a character string on the Message line.



string is a quoted string that contains message text.

variable is a CHAR or VARCHAR variable that contains message text.

Usage

You can specify any combination of variables and strings for the message text. 4GL generates the message to display by replacing any variables with their values and concatenating the strings. If the length of the message text exceeds the width of the screen or 4GL window, the text is truncated to fit.

The Message Line

4GL displays message text in the Message line. 4GL positions this reserved line according to default or explicit Message line specification for the program or for the current 4GL window, in this order of descending precedence:

1. A MESSAGE LINE specification in the most recent OPTIONS statement.
2. A MESSAGE LINE specified in the most recent OPEN WINDOW statement.
3. The default Message line, or the *second* line of the current 4GL window.

The message remains on the screen until you display a menu or another message. To clear the Message line, you can display a blank message, like this:

```
MESSAGE " "
```

You can include the CLIPPED and USING operators in a MESSAGE statement. For example, the following MESSAGE statement uses the CLIPPED operator to remove any trailing blanks from the string in the variable **file_name**:

```
DEFINE file_name CHAR(20)
...
MESSAGE "Printing mailing labels to", file_name CLIPPED,
    " -- Please wait"
```

You can also use the ASCII and COLUMN operators. For information on using the 4GL built-in functions and operators, see Chapter 4.

If you position the Message line so that it coincides with the Comment line, Menu line, or fields of a form, then output from the MESSAGE statement is not visible. For example:

```
DATABASE stores
MAIN
    DEFINE p_customer RECORD LIKE customer.*
    OPEN WINDOW r1 AT 4,1 WITH FORM "platonic"
        ATTRIBUTE (MESSAGE LINE LAST)
    MESSAGE "This is a word to the wise."
    INPUT BY NAME p_customer.*
    CLOSE WINDOW r1
END MAIN
```

This program does not display the text of the MESSAGE statement, because the default position of the Comment line is also the last line. If the ATTRIBUTE clause of the OPEN WINDOW statement in the same example were revised to specify

```
ATTRIBUTE (MESSAGE LINE LAST, COMMENT LINE FIRST)
```

so that there is no conflict between those reserved lines, then the message text will appear when the MESSAGE statement is executed. The sections "Positioning Reserved Lines" (page 3-225 and 3-231) describe the syntax of the OPEN WINDOW and OPTIONS statements to position these reserved lines.

The ATTRIBUTE Clause

For general information about the syntax and use of the ATTRIBUTE clause, see page 3-290. This section describes specific information about using the ATTRIBUTE clause within a MESSAGE statement.

The default display attribute for the Message line is the NORMAL display. You can use the ATTRIBUTE clause to alter the default display attribute of the Message line. For example, the following statement changes the display attribute of the message text to reverse video:

```
MESSAGE "Please enter a value " ATTRIBUTE (REVERSE)
```

4GL ignores the INVISIBLE attribute if you include it in the ATTRIBUTE clause of the MESSAGE statement.

You can refer to substrings of CHAR, VARCHAR, and TEXT type variables by following the variable name with a pair of integers to indicate the starting and ending position of the substring, enclosed between brackets ([ ]) and separated by a comma (,) symbol. For example, the following MESSAGE statement displays a 10-character substring of the **full_name** variable:

```
MESSAGE "Customer ", full_name[11,20]
    CLIPPED, " added to the database"
```

Statements in the next program fragment perform the following tasks:

1. Uses a MESSAGE statement to clear the Message line of any text.
2. Clears all the fields of the current form.
3. Uses a PROMPT statement to instruct the user to type a name.
4. Assigns the value of the entered string to the variable **last_name**.
5. Uses another MESSAGE statement to indicate to the user that the program is retrieving rows.
6. Clears the second message after a 3-second delay.

```
MESSAGE ""
CLEAR FORM
PROMPT "Enter a last name:" FOR last_name
MESSAGE "Selecting rows for customer with last name ",
    last_name, ". . ." ATTRIBUTE (YELLOW)
SLEEP 3
MESSAGE ""
```

References

DISPLAY, ERROR, OPEN WINDOW, OPTIONS, PROMPT

NEED

NEED is a conditional statement to control output from the PRINT statement. (The NEED statement can appear only in a REPORT program block.)

NEED *lines* LINES

lines is an integer expression (page 3-338) that specifies how many lines must remain in the current page between the line above the current character position and the bottom margin.

Usage

The NEED statement causes subsequent report output from the PRINT statement to start on the next page of the report, if fewer than the specified number of available lines remain between the current line of the page and the bottom margin. NEED has the effect of a conditional SKIP TO TOP OF PAGE, the condition being that the number returned by the integer expression must be greater than the number of lines that remain on the current page.

The NEED statement can prevent INFORMIX-4GL from separating parts of the report that you want to keep together on a single page. In the following example, the NEED statement causes the PRINT statement to send output to the next page, unless at lease six lines remain on the current page:

```
AFTER GROUP OF r.order_num
    NEED 6 LINES
    PRINT " ",r.order_date, 7 SPACES,
        GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

NEED does not include the BOTTOM MARGIN value in calculating the lines available. If the number of lines remaining above the bottom margin on the page is less than *lines*, then both the PAGE TRAILER and the PAGE HEADER are printed before the next PRINT statement is executed. You cannot include the NEED statement in the PAGE HEADER nor PAGE TRAILER control blocks.

References

PAUSE, PRINT, REPORT, SKIP

OPEN FORM

The OPEN FORM statement declares the name of a compiled 4GL form.

OPEN FORM *form* FROM "*filename*"

filename is a quoted string that specifies the name of a file that contains the compiled screen form. This can also include a pathname.

form is a 4GL identifier that you assign here as the name of the form.

Usage

To display a form, you can follow these steps:

1. Create a form specification file. (This file should have a **.per** extension.)
2. Compile the form by using the Compile option of the Form menu in the Programmer's Environment or by using the **form4gl** command. (The compiled form file has **.frm** as its file extension.)
3. Declare the form name by using the OPEN FORM statement.
4. Display the form by using the DISPLAY FORM statement.

Once 4GL displays the form, you can activate the form by executing the CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY statements.

When it executes the OPEN FORM statement, 4GL loads the compiled form into memory. (The CLOSE FORM statement is a memory-management feature to recover memory from forms that 4GL no longer displays on the screen.)

The Form Name

The *form* name need not match the name of the form specification file, but it must be unique among form names in the program. Its scope of reference is the entire program. page 2-9 describes the rules for 4GL identifiers.

Specifying a Filename

The quoted string that follows the FROM keyword must specify the name of the file that contains the compiled screen form. This *filename* can include a pathname. You can omit or include the **.frm** extension:

```
OPEN FORM frmofmor FROM "/fomr/fmro.frm"
```

Displaying a Form in a 4GL Window

To position the form in a 4GL window, precede the OPEN FORM statement with the OPEN WINDOW statement. The following program fragment opens the **w_cust1** window, opens and displays the **o_cust** form in that 4GL window, and calls the **cust_order( )** function. When the function returns, the CLOSE WINDOW statement closes both the form and the 4GL window:

```
MAIN
    OPEN WINDOW w_cust1 AT 10,15
        WITH 11 ROWS, 63 COLUMNS
        ATTRIBUTE (BORDER)
    OPEN  FORM  o_cust  FROM  "custorder"
    DISPLAY FORM o_cust
    CALL cust_order()
    CLOSE WINDOW w_cust1
END MAIN
```

If you execute an OPEN FORM statement with the name of an open form, 4GL first closes the existing form before opening the new form.

The WITH FORM keywords of OPEN FORM both open and display a form in a 4GL window. You do not need to execute the OPEN FORM, DISPLAY FORM, and CLOSE FORM statements if you use the OPEN WINDOW statement to display the form. You also do not need to use the CLOSE FORM statement to release the memory allocated to the form. Instead, you can use the CLOSE WINDOW statement to close both the form and the 4GL window, and to release the memory. For example, the following statements open 4GL window **w_cust2**, call function **cust_order( )**, and then close the 4GL window:

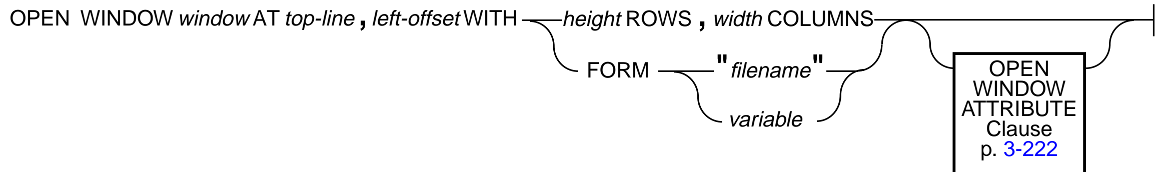
```
OPEN WINDOW w_cust2 AT 10,15 WITH FORM "custorder"
CALL cust_order()
CLOSE WINDOW w_cust2
```

References

CLEAR, CLOSE FORM, CLOSE WINDOW, CURRENT WINDOW, OPEN WINDOW, OPTIONS

OPEN WINDOW

The OPEN WINDOW statement declares and opens a 4GL window.



filename is a quoted string that specifies the file containing a compiled 4GL form. This can include a drive, pathname, and file extension.

height is an integer expression (page 3-338) to specify the height, in lines.

left-offset is an integer expression to specify the left margin, in characters, where 0 = the left edge of the 4GL screen.

top-line is an integer expression to specify the position of the top line of the 4GL window, where 0 = the top of the 4GL screen.

variable is a CHAR or VARCHAR variable that specifies the *filename*.

width is an integer expression to specify the width, in characters.

window is a name that you declare here for the 4GL window to be opened.

Usage

A 4GL *window* is a rectangular area in the 4GL screen that can display a form, a menu, or output from the DISPLAY, MESSAGE, or PROMPT statements. Note that the term "4GL window" is distinct from other windows, such as the Help window. (That is, the 4GL screen can display one or more *4GL windows*.) An OPEN WINDOW statement can have the following effects:

- Declares a name for the 4GL window.
- Specifies the position of the 4GL window on the 4GL screen.
- Defines the dimensions of the 4GL window, in lines and characters.
- Specifies the display attributes of the 4GL window.

The *window* identifier must follow the rules for 4GL identifiers (page 2-9) and be unique among 4GL windows in the program. Its scope is the entire program. You can use this identifier to reference the same 4GL window in other statements (for example, CLEAR, CURRENT WINDOW, and CLOSE WINDOW).

The following topics are described in this section:

Topic	Page
The 4GL Window Stack	3-220
The AT Clause	3-220
The WITH ROWS, COLUMNS Clause	3-221
The WITH FORM Clause	3-221
The OPEN WINDOW ATTRIBUTE Clause	3-222
The Color and Intensity Attributes	3-224
The BORDER Attribute	3-224
Positioning Reserved Lines	3-225

The 4GL Window Stack

INFORMIX-4GL maintains a *window stack* of all open 4GL windows. If you execute OPEN WINDOW to open a new 4GL window, 4GL does the following:

- Saves any changes made to the current 4GL window.
- Adds the new 4GL window to the window stack.
- Makes the new 4GL window the current 4GL window.

Other statements that can modify the window stack are CURRENT WINDOW (page 3-56) and CLOSE WINDOW (page 3-30).

The AT Clause

The AT clause specifies the location of the top left corner of the 4GL window. The location is relative to the entire 4GL screen, and independent of any other 4GL windows. You must specify these coordinates as expressions that return positive integers within the following ranges:

- The first expression must return an integer between 1 and (*max* - *lines*), for *max* the maximum number of lines in the 4GL screen, and *lines* the ROWS specification (page 3-221). The 4GL window begins on this line.
- The second expression must return a whole number between 1 and (*length* - *characters*), where *length* is the maximum number of characters that the 4GL screen can display on one line, and *characters* is the COLUMNS specification (page 3-221). This is the left margin.

A comma separates the two expressions in the AT clause. For example, the following statement opens a 4GL window with the top left corner at the third line and the fifth character position of the 4GL screen:

```
OPEN WINDOW o1 AT LENGTH("Mom"), 5 WITH 10 ROWS, 40 COLUMNS
```

The WITH ROWS, COLUMNS Clause

The WITH *lines* ROWS, *characters* COLUMNS clause specifies explicit vertical and horizontal dimensions for the 4GL window.

- The expression at the left of the ROWS keyword specifies the height of the 4GL window, in *lines*. This must be an integer between 1 and *max*, for *max* the maximum number of lines that the 4GL screen can display.
- The integer expression after the comma at the left of the COLUMNS keyword specifies the width of the 4GL window, in *characters*. This must return a whole number between 1 and *length*, for *length* the number of characters that your monitor can display on one line.

This statement opens a 4GL window 5 lines high and 74 characters wide:

```
OPEN WINDOW w2 AT 10, 12 WITH 5 ROWS, 74 COLUMNS
```

In addition to the lines needed for a form, allow room for reserved lines:

- The Comment line. (By default, this is the last line of the 4GL window.)
- The Form line (By default, this is line 3 of the 4GL window.)
- The Error line. (By default, this is the last line of the 4GL screen, not of the 4GL window).

4GL issues a run-time error if the 4GL window does not include sufficient lines in to display both the form and these additional reserved lines.
To reduce the number of lines required by 4GL, you can define the Form line as line one or two, and change other reserved lines accordingly, such as the Prompt and Menu lines. For information on how to make these changes, see the section "The OPEN WINDOW ATTRIBUTE Clause" on page 3-222.

The minimum number of lines required to display a form in a 4GL window is the number of *lines* in the form, plus an additional line below the form for prompts, messages, and comments.

The WITH FORM Clause

As an alternative to specifying explicit dimensions, the WITH FORM clause can specify a quoted string or a character variable that specifies the name of a file that contains the compiled screen form. You can omit or include the **.frm** file extension. 4GL automatically opens a 4GL window sized to the screen layout (page 5-14) of the form, and displays the form.

If you include a WITH FORM clause, the width of the 4GL window is from the left-most character on the screen form (including leading blank spaces) to the right-most character on the screen form (truncating trailing blank spaces).

The length of the 4GL window is the following sum:

(*form line*) + (*form length*)

Here *form line* is the reserved line position on which to display the first line of the form (by default, line 3) and *form length* is the number of lines in the screen layout of the SCREEN section of the form specification file. 4GL adds one line for the Comment line. Unless you specify FORM LINE in an ATTRIBUTE clause or in the OPTIONS statement, the default value of this sum is *form length* + 2. (For more information on screen layouts in 4GL forms, see page 5-12.)

For example, the following statement opens a 4GL window called **w1** and positions its top left corner at the fifth row and fifth column of the 4GL screen. The WITH FORM clause opens and displays the **custform** form in this 4GL window. If **custform** were 10 lines long and the FORM LINE option were the default value (3), the height of **w1** would be (10 + 3) = 13 lines:

```
OPEN WINDOW w1 AT 5, 5 WITH FORM "custform"
```

The WITH FORM clause is convenient if the 4GL window always displays the same form. If you use this clause, you do not need the OPEN FORM, DISPLAY FORM, nor CLOSE FORM statements to open and close the form:

- The OPEN WINDOW WITH FORM statement opens and displays the form.
- The CLOSE WINDOW statement closes the 4GL window and the form.

You *cannot* use the WITH FORM clause in the following cases:

- To display more than one form in the same 4GL window.
- To display a 4GL window larger than the default dimensions that 4GL supplies (as described above) when it executes the WITH FORM clause.

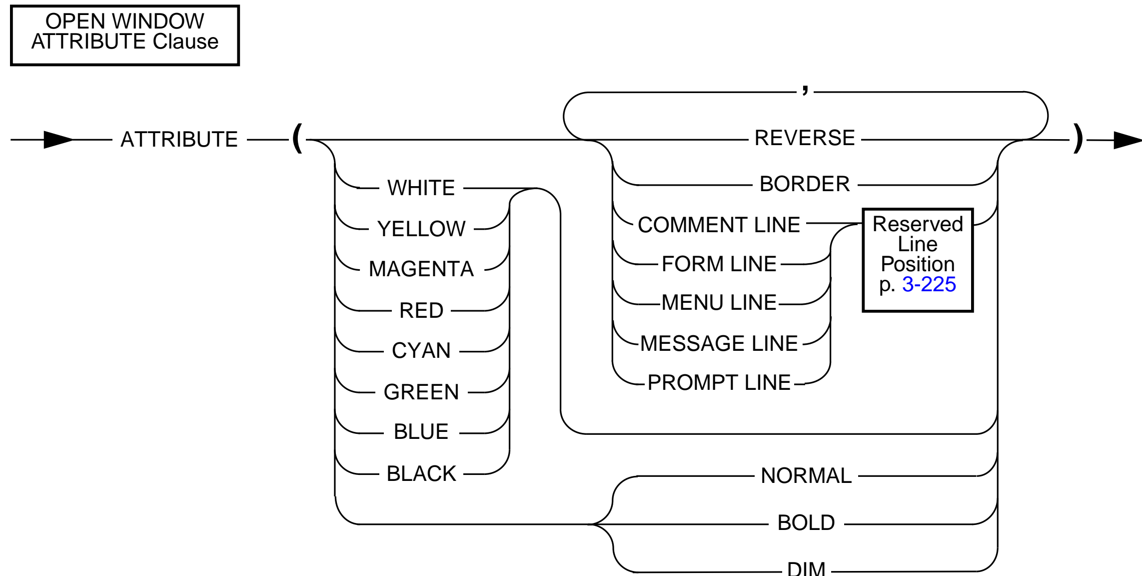
In these cases, you must specify explicit dimensions by using the WITH *lines* ROWS, *characters* COLUMNS clause. You must also execute the OPEN FORM, DISPLAY FORM, and CLOSE FORM statements to open, display, and close the form or forms explicitly. (You typically are not required to use the CLOSE FORM statement, which affects memory management, rather than the visual interface of your program.)

The OPEN WINDOW ATTRIBUTE Clause

Use the OPEN WINDOW ATTRIBUTE clause to do the following:

- Specify a border for the 4GL window.
- Display the 4GL window in reverse video or in a color.
- Reposition the Prompt, Message, Menu, Form, and Comment lines.

The OPEN WINDOW ATTRIBUTE clause has the following syntax:



The *color* attributes are listed in the left-hand portion of the diagram. Besides these, you can also specify INVISIBLE as a *color*, but this specification has no effect in the OPEN WINDOW ATTRIBUTE clause. Without this clause, the attributes and reserved line positions have the following default values:

Attribute	Default Setting
color	The default foreground color on your terminal
REVERSE	No reverse video
BORDER	No border
PROMPT LINE *line valuee*	FIRST (=1)
MESSAGE LINE *line value*	FIRST + 1 (=2)
MENU LINE *line value*	FIRST (=1)
FORM LINE *line value*	FIRST + 2 (=3)
COMMENT LINE *line value*	LAST - 1 (for the **4GL** screen) LAST (for all other **4GL** windows)

For more information on valid *reserved line* values, see page 3-225. For more information about color and intensity attributes, see page 3-290.

If you specify a color or the REVERSE attribute in the ATTRIBUTE clause of an OPEN WINDOW statement, it becomes the default attribute for displays in the 4GL window, except for menus. You can override this default by specifying a different attribute in the ATTRIBUTE clause of the CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, INPUT or INPUT ARRAY statement.

The Color and Intensity Attributes

Display attributes can be classified as *color* and *intensity* (or *monochrome)* attributes. The color attributes (page 3-222) override the default foreground color on your teminal. On monochrome monitors, all color attributes except BLACK are displayed as WHITE.

4GL displays the intensity attributes as follows on color monitors:

Attribute	Displayed As
NORMAL	WHITE
BOLD	RED
DIM	BLUE

For example, if you have a color monitor, the 4GL window specified in the following statement is displayed with the BLUE attribute:

```
OPEN WINDOW w2 AT 10, 12 WITH 5 ROWS, 40 COLUMNS ATTRIBUTE (BLUE)
```

On a monochrome display, the BLUE attribute produces a *white* 4GL window.

The REVERSE Attribute

Use the REVERSE attribute to display the foreground of the 4GL window in reverse video (sometimes called "inverse video"). The following statement assigns the BLUE and REVERSE attributes to the **w2** window:

```
OPEN  WINDOW w2 AT 10, 12 WITH 5 ROWS, 40 COLUMNS
    ATTRIBUTE (BLUE,REVERSE)
```

The BORDER Attribute

The BORDER attribute draws a border *outside* the specified 4GL window. The border requires two lines on the screen (one above and another below the window) and two character positions (one to the left and one to the right of the window). Make sure to account for this space when you specify coordinates in the AT clause. For example, the following statement opens a 4GL window and displays a border around it:

```
OPEN WINDOW w1 AT 10,10 WITH 5 ROWS, 30 COLUMNS ATTRIBUTE (BORDER)
```

The following diagram indicates the coordinates of the border enclosing the 5x30 4GL window that was specified in the preceding example:

Note that the coordinates of the top left corner of the window border are 9, 9. The 4GL window itself starts at 10, 10.

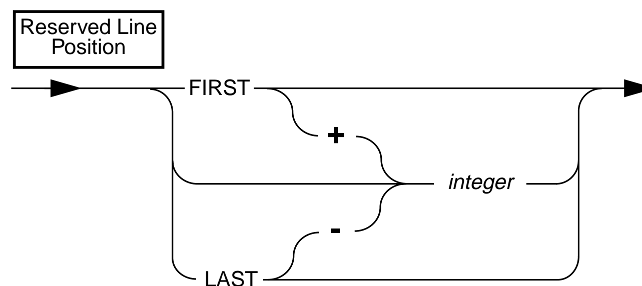
4GL draws the window with the characters defined in the **termcap** or **terminfo** files. You can specify alternative characters in these files. Otherwise, 4GL uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners. Some **termcap** or **terminfo** files have settings that require additional row and columns to display windows. For more information, see Appendix F.

If a window and its border exceed the physical limits of the screen, a run-time error occurs.

See also the built-in FGL_DRAWBOX( ) function that displays rectangles (page 4-56).

Positioning Reserved Lines

The Reserved Line Position segment has the following syntax:



Line values specified in the OPTIONS ATTRIBUTE clause (page 3-231) of the most recently executed OPTIONS statement can position the Form, Prompt, Menu, Message, Comment, and Error lines. If no line positions are specified in the OPTIONS ATTRIBUTE nor OPEN WINDOW ATTRIBUTE clause, then the 4GL window uses the following default positions for its reserved lines:

Default Location	Reserved for
First line	Prompt line (output from PROMPT statement); *also* Menu line (*command value* from MENU statement)
Second line	Message line (output from MESSAGE statement; *also* the *description value* output from MENU statement)
Third line	Form line (output from DISPLAY FORM statement)
Last line	Comment line in any **4GL** window except SCREEN

These positional values are relative to the first or last line of the 4GL window, rather than to the 4GL screen. (The Error line is always the last line of the 4GL screen.) When you open a new 4GL window, however, the OPEN WINDOW ATTRIBUTE clause can override these defaults for every reserved line (except the Error line). This disables the OPTIONS statement reserved line specifications only for the specified 4GL window.

Except for the cases that are described in notes that follow, the *position* that you specify for the reserved lines of 4GL can be any of the following:

- FIRST
- FIRST + *integer*
- *integer*
- LAST - *integer*
- LAST

  Here *integer* is a literal or variable that returns a positive whole number, such that the LINE specification is no greater than the number of lines in the 4GL window. This is true for all reserved lines except the following:

- The Menu line: do not specify LAST.

  A menu requires two lines. The menu *title* and *commands* appear on the Menu line, and *command description* appears on the next line. To display a menu at the bottom of a 4GL window, specify MENU LINE LAST - 1.

- The Form line: do not specify LAST nor LAST - *integer*.

Here FIRST is the first line of the 4GL window (line 1), and LAST is the last line. The following statement sets three reserved line positions:

```
OPEN WINDOW wcust AT 3,6 WITH 10 ROWS, 50 COLUMNS
ATTRIBUTE (MESSAGE LINE 20,
        PROMPT LINE LAST-2,
        FORM LINE FIRST)
```

If a 4GL window is not large enough to contain the specified value for one or more of these reserved lines, 4GL increases its line value to FIRST or decreases it to LAST, whichever may be appropriate.

If the 4GL window is not wide enough to display all the text that you specify, 4GL truncates the message. You can use these features to display text:

- PROMPT statement
- MESSAGE statement
- DISPLAY statement
- COMMENTS attribute of a screen form

Because the position of the Error line is relative to the 4GL screen, rather than to the current 4GL window, the ATTRIBUTE clause of an OPEN WINDOW statement cannot change the location of the Error line. Use the OPTIONS statement (page 3-229) to change the position of the Error line.

Because the INPUT statement clears both the Comment line and the Error line when moving between fields, it is not a good idea to set the Message or Prompt line to any of the following:

- The last line of the 4GL window. (This is the default Comment line.)
- The last line of the 4GL screen. (This is the default Error line.)

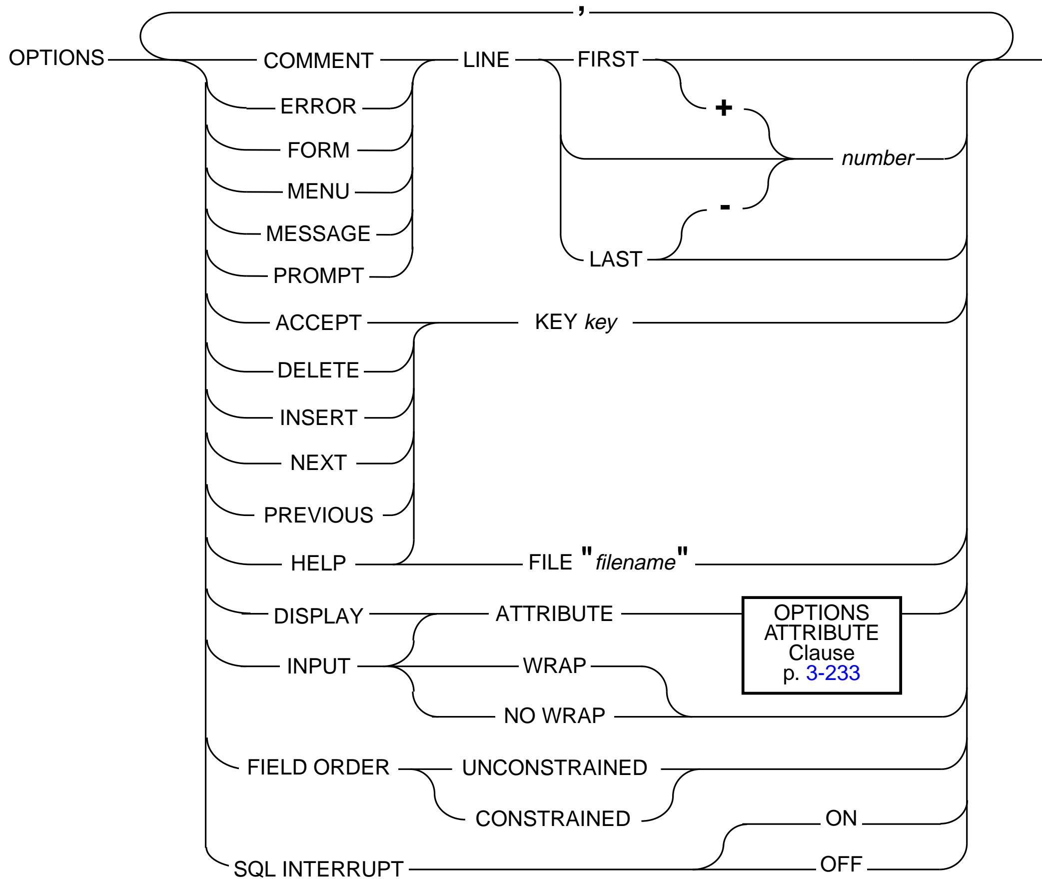
To use these lines for messages or prompts, be sure to redefine the Comment and Error lines too.

References

CLEAR, CLOSE FORM, CLOSE WINDOW, CURRENT WINDOW, DISPLAY, MESSAGE, OPEN FORM, OPTIONS, PROMPT

OPTIONS

The OPTIONS statement specifies default features of form-related statements and other 4GL screen interaction statements.



filename is a quoted string that specifies the name of a file that contains the compiled Help messages. This can also include a pathname.

key is a keyword to specify a physical or logical key (page 3-234).

number is a literal integer (page 3-340) to specify a line number.

Usage

The OPTIONS statements sets defaults for screen-interaction statements.

The following topics are described in sections that follow:

Topic	Page
Features Controlled by OPTIONS Clauses	3-229
Positioning Reserved Lines	3-231
Cursor Movement in Interactive Statements	3-232
The OPTIONS ATTRIBUTE Clause	3-233
The HELP FILE Option	3-234
Assigning Logical Keys	3-234
Interrupting SQL Statements	3-235

Features Controlled by OPTIONS Clauses

A program can include several OPTIONS statements. If these conflict in their specifications, the OPTIONS statement most recently encountered at run-time prevails. OPTIONS can specify various features of form-related statements, including CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MESSAGE, OPEN FORM, OPEN WINDOW, and PROMPT:

- Positions of the *reserved lines* of 4GL
- Input and display *attributes*
- *Logical key* assignments
- The name of the *Help file*
- Whether SQL statements can be *interrupted*
- *Field traversal* constraints

If you omit the OPTIONS statement, 4GL uses defaults that are described here:

Clause	Effect
COMMENT LINE	Specifies the position of the Comment line. This displays messages defined with the COMMENT attribute in the form specification file. The default is (LAST - 1) for the 4GL screen, and LAST for all other **4GL** windows.
ERROR LINE	Specifies the position in the **4GL** screen of the Error line that displays text from the ERROR statement. The default is the LAST line of the **4GL** screen.
FORM LINE	Specifies the position of the first line of a form. The default is (FIRST + 2), or line 3 of the current **4GL** window.
MENU LINE	Specifies the position of the Menu line. This displays the menu name and options, as defined by the MENU statement. The default is the FIRST line of the **4GL** window.

MESSAGE LINE	Specifies the position of the Message line. This reserved line displays the text listed in the MESSAGE statement. The default is (FIRST + 1), or line 2 of the current **4GL** window.
PROMPT LINE	Specifies the position of the Prompt line, to display text from PROMPT statements. The default value is the FIRST line of the **4GL** window.
ACCEPT KEY	Specifies the key to terminate an CONSTRUCT, INPUT, INPUT ARRAY, or DISPLAY ARRAY. Default is ESCAPE.
DELETE KEY	Specifies the key in INPUT ARRAY statements to delete a screen record. The default Delete key is F2.
INSERT KEY	Specifies the key to open a screen record for data entry in INPUT ARRAY. The default Insert key is F1.
NEXT KEY	Specifies the key to scroll to the next page of a program array of records in an INPUT ARRAY or DISPLAY ARRAY statement. The default Next key is F3.
PREVIOUS KEY	Specifies the key to scroll to the previous page of program records in an INPUT ARRAY or DISPLAY ARRAY statement. The default Previous key is F4.
HELP KEY	Specifies the key to display Help messages. The default Help key is CONTROL-W.
HELP FILE	Specifies the file (produced by the **mkmessage** utility) containing programmer-defined Help messages.
DISPLAY ATTRIBUTE	Specifies default attributes to use during DISPLAY or DISPLAY ARRAY statement when none are specified by those statements or in the form specification file.
INPUT ATTRIBUTE	Specifies the attributes to use during a CONSTRUCT or INPUT statement when no attributes are specified by those statements or in the form specification file.
INPUT NO WRAP	Specifies that the cursor does not "wrap." An INPUT or CONSTRUCT statement terminates when a user presses RETURN after the last field. This is the default value.
INPUT WRAP	Specifies that the cursor "wraps" between the last and first input fields during INPUT, INPUT ARRAY, or CONSTRUCT statements, until the user presses the Accept key. Pressing RETURN at the last field does not deactivate the form.
FIELD ORDER CONSTRAINED	Specifies that the Up arrow key moves the cursor to the previous field and the Down arrow key moves the cursor to the next field when users enter values for CONSTRUCT or INPUT statements.
FIELD ORDER UNCONSTRAINED	Specifies that the Up arrow key moves the cursor to the field above the current position and the Down arrow key moves the cursor to the field below the current cursor position when users enter values for CONSTRUCT or INPUT statements.

SQL INTERRUPT ON	Specifies that the user can interrupt SQL statements as well as **4GL** statements.
SQL INTERRUPT OFF	Specifies that the user cannot interrupt SQL statements.

Positioning Reserved Lines

Except for the cases that are described below, the *position* that you specify for each reserved line of 4GL can be any of the following:

- FIRST
- FIRST + *integer*
- *integer*
- LAST - *integer*
- LAST
  Here *integer* is a variable or a literal that returns a positive whole number, such that the LINE specification is no greater than the number of lines in the 4GL window or 4GL screen, except for these reserved lines:
- The Form line: do not specify LAST nor LAST - *integer*.
- The Menu line: do not specify LAST. A ring menu requires two lines. The menu *title* and *commands* appear on the Menu line, and the *command description* appears on the following line. If you want a menu to appear at the bottom of a 4GL window, specify MENU LINE LAST - 1.

FIRST is the top line of the current 4GL window (line 1), and LAST is the last line. For example, the following statement sets three reserved line positions:

```
OPTIONS MENU LINE 20, PROMPT LINE LAST-2, FORM LINE FIRST
```

If a 4GL window is not large enough to contain the specified value for one or more of these reserved lines, 4GL automatically increases its line value to FIRST or decreases it to LAST, as appropriate.

The line position for the Error line is relative to the 4GL screen, rather than to the current 4GL window. The line value of any other reserved line is relative to the first line of the current 4GL window (or to the 4GL screen, if that is the current 4GL window). If the 4GL window is not wide enough to display all the message text that you specify, then 4GL truncates the message. You can use these features of 4GL to display message text:

- PROMPT statement
- MESSAGE statement
- DISPLAY statement
- ERROR statement

- COMMENTS attribute of a form specification file

Because the INPUT statement clears both the Comment line and the Error line when the cursor moves between fields, it is not a good idea to set the Message line or the Prompt line to either of the following positions:

- The last line of the current 4GL window; this is the default Comment line.
- The last line of the 4GL screen; this is the default Error line.

Default line positions set by OPTIONS remain in effect until another OPTIONS statement redefines them. They can also be reset by the ATTRIBUTE clause of the OPEN WINDOW statement (page 3-225), but only for the specified 4GL window; after it closes, the reserved line positions are restored to their values from the most recently executed OPTIONS statement.

Cursor Movement in Interactive Statements

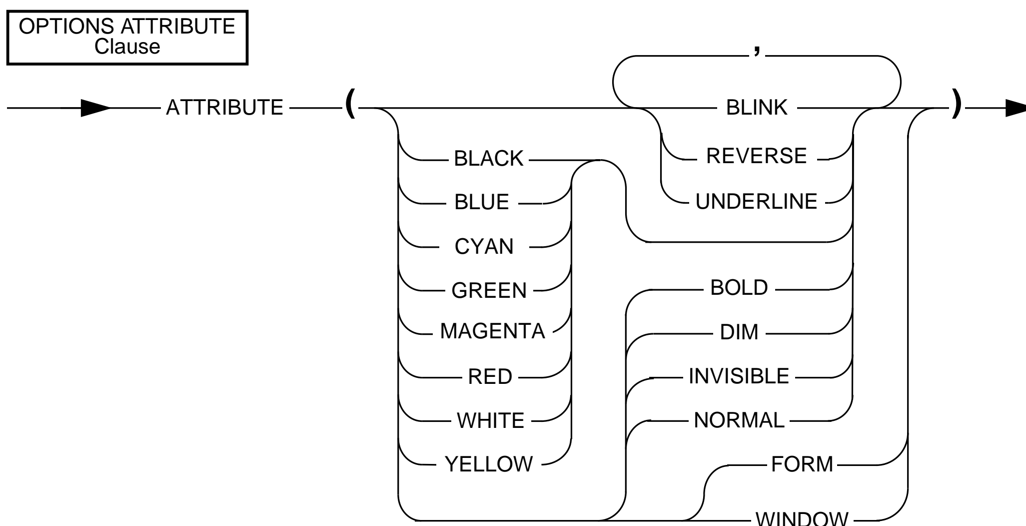
The TAB order in which the screen cursor visits fields of a form is that of the *field list* of currently executing CONSTRUCT, INPUT, or INPUT ARRAY statements, except as modified by NEXT FIELD clause. By default, the interactive statement terminates if the user presses the RETURN key in the last field (or if entered data fills the last field, if that field has the AUTONEXT attribute).

The INPUT WRAP keywords change this, causing the cursor to move from the last field to the first, repeating the sequence of fields until the user presses the Accept key. The INPUT NO WRAP option restores the default cursor behavior.

Specify FIELD ORDER UNCONSTRAINED to cause the Up and Down arrow keys to move the cursor to the field above or below, respectively. Use the FIELD ORDER CONSTRAINED option to restore the default behavior of the Up and Down arrow keys moving the cursor to the previous or next field, respectively.

The OPTIONS ATTRIBUTE Clause

This section describes the OPTIONS ATTRIBUTE clause. It explains the FORM keyword and WINDOW keyword in detail. For generic information about the ATTRIBUTE clause, see page 3-290.



This clause can specify features for input statements (CONSTRUCT, INPUT, and INPUT ARRAY) and for display statements (DISPLAY and DISPLAY ARRAY):

- The attributes of the foreground of the 4GL window.
- Whether to use input attributes of the current form or 4GL window.
- Whether to use display attributes of the current form or 4GL window.

If this clause conflicts with another attribute specification, 4GL applies the precedence rules that are listed on page 3-292. Any attribute defined by the OPTIONS statement remains in effect until 4GL encounters an ATTRIBUTES clause that redefines the same attribute in one of the following statements:

- CONSTRUCT, INPUT, INPUT ARRAY, DISPLAY, or DISPLAY ARRAY
- Another OPTIONS statement
- An OPEN WINDOW statement

An ATTRIBUTE clause of an OPEN WINDOW, CONSTRUCT, INPUT, DISPLAY, or DISPLAY ARRAY statement only temporarily redefines the attributes. After the 4GL window closes (in the case of an OPEN WINDOW statement) or after the statement terminates (in the case of an input or display statement), 4GL restores the attributes from the most recent OPTIONS statement.

The FORM keyword in INPUT ATTRIBUTE or DISPLAY ATTRIBUTE clauses instructs 4GL to use the input or display attributes of the current form. For example, this uses the display attributes from the form specification file:

```
OPTIONS DISPLAY ATTRIBUTE (FORM)
```

Similarly, you can use the WINDOW keyword of the same options to instruct 4GL to use the input or display attributes of the current 4GL window. You cannot combine the FORM or WINDOW attributes with any other attributes.

The HELP FILE Option

The HELP FILE clause specifies an expression that returns the *filename* of a Help file. This *filename* can also include a pathname.

Messages in this file can be referenced by number in form-related statements, and are displayed at run time when the user chooses the Help key. (The **mkmessage** utility for Help files is described in Appendix B.)

Assigning Logical Keys

The OPTIONS statement can specify physical keys to support 4GL logical key functions in the current task. You can specify the following keywords in uppercase or lowercase characters for *key name*:

DOWN	NEXT *or* NEXTPAGE	TAB
ESC *or* ESCAPE	PREVIOUS *or* PREVPAGE	UP
INTERRUPT	RETURN *or* ENTER	
LEFT	RIGHT	

F1 *through* F64

CONTROL-*char* (for *char* any letter except A, D, H, I, J, L, M, R, or X)

For example, this statement redefines the Next Page and Previous Page keys:

```
OPTIONS NEXT KEY CONTROL-N, PREVIOUS KEY CONTROL-P
```

The keyword NEXTPAGE is a synonym for NEXT in 4GL statements (like CONSTRUCT, DISPLAY ARRAY, INPUT, MENU, OPTIONS, and PROMPT) that reference the Next Page key. Similarly, the keyword PREVPAGE is a synonym for PREVIOUS in statements that reference the Previous Page key.

The following table lists keys that require special consideration before you assign them in an OPTIONS statement:

Key	Special Considerations
ESC or ESCAPE	You must specify another key as the Accept key because ESCAPE is the default Accept key. Reassign the Accept key in the OPTIONS statement.
INTERRUPT	You must first execute a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, **4GL** executes the statements in the ON KEY block and sets the global variable **int_flag** to nonzero, but does not terminate the current statement. **4GL** also executes the ON KEY statement block if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, **4GL** sets the **quit_flag** variable for the current task to non-zero.
CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these control keys for field editing.
I, J, and M	The standard meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is lost to the user. Instead, the key is "trapped" by 4GL and used to trigger the commands in the OPTIONS statement. For example, if CONTROL-M appears in an OPTIONS statement, the user cannot press RETURN to advance the cursor to the next field. If you include one of these keys in an OPTIONS statement, also restrict the scope of the statement.

You may not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

To disable a key function, you can assign it to a control sequence that will never be executed. For example, the editing control sequences (CONTROL-A, -D, -H, -L, -R, and -X) are always interpreted as field editing commands. If you assign one of these control sequences to a key function, 4GL executes the editing sequence instead of the key function. For example, the following statement disables the Delete key:

```
OPTIONS DELETE KEY CONTROL-A
```

After 4GL processes this statement, the user is no longer able to delete rows in a screen array.

Interrupting SQL Statements

The SQL INTERRUPT option specifies whether the Interrupt key interrupts SQL statements as well as **4GL** statements. By default, this option is set to OFF so pressing the Interrupt key cannot interrupt SQL statements. If the user

presses the Interrupt key when an SQL statement is executing. **4GL** waits for the database engine to complete the SQL statement before processing the Interrupt:

- If the program contains the DEFER INTERRUPT statement, **4GL** sets the **int_flag** built-in variable to TRUE and continues execution.
- If the program does not contain DEFER INTERRUPT, **4GL** terminates the program.

For more information on the actions of the DEFER INTERRUPT statement, see the DEFER statement on page 3-62.

To enable the Interrupt key to interrupt SQL statements, your program must contain:

- The DEFER INTERRUPT statement.
- The OPTIONS statement with the SQL INTERRUPT ON option.

When your program contains both these statements, **4GL** takes the following actions when the user presses the Interrupt key:

1. Tells the database engine to terminate the current SQL statement. SQL statements which can be terminated include:

SQL Statement	Considerations
ALTER INDEX	Can be interrupted by INFORMIX-OnLine engine only
ALTER TABLE	
CREATE INDEX	Can be interrupted by INFORMIX-OnLine engine only
DELETE	
FETCH	Includes implicit FETCH performed during a FOREACH
INSERT	Includes INSERT performed during a LOAD
OPEN	If the SELECT stores all the data in a temporary table
SELECT	Includes SELECT performed during an UNLOAD
UPDATE	

   If the interrupted SQL statement is within a database transaction, then the database engine must handle the interrupted transaction. See "Interrupting Transactions" on page 3-237 for more information.

2. Sets the built-in **int_flag** to TRUE.
3. Sets the global SQLCA.SQLCODE and **status** variables to an error code of -213.

4. Continues execution with the statement following the interrupted SQL statement, if your program has the WHENEVER ERROR CONTINUE compiler directive in effect; otherwise, the program terminates.

For SQL statements not listed in the table above, **4GL** will allow the statement to complete before setting the built-in **int_flag** variable. It will then continue execution with the statement following the SQL statement (if your program has the WHENEVER ERROR CONTINUE compiler directive in effect).

If the DEFER QUIT statement has been executed and the user presses the Quit key (or sends a SIGQUIT signal), then 4GL takes the same four actions, except that it sets the global variable **quit_flag**, rather than **int_flag**.

If you specify SQL INTERRUPT ON, but later in the program you wish to disable the SQL interruption feature, execute an OPTIONS SQL INTERRUPT OFF statement. This restores the default of uninterruptable SQL statements.

Interrupting Transactions

Interrupting an SQL statement has consequences for database transactions. In typical 4GL applications, the SQL INTERRUPT ON feature is of very limited value unless the database supports transaction logging.

How to handle an interrupted SQL statement depends upon whether the database is ANSI-compliant and on what type of transaction the SQL statement is within:

- Database which is *not* ANSI-compliant—if the database supports transaction logging, a transaction is either:
  - An *explicit* transaction—started with the BEGIN WORK statement and ended with either the COMMIT WORK (save the transaction) or ROLLBACK WORK (cancel the transaction) statement.
  - A *singleton* transaction—an SQL statement that is not within an explicit transaction (preceded by a BEGIN WORK) is in a transaction of its own. The transaction ends when the SQL statement completes.
- ANSI-compliant database—has *implicit* transactions.

  A transaction is always in effect. The BEGIN WORK statement is not needed because every SQL statement is automatically within a transaction. A COMMIT WORK or ROLLBACK WORK statement automatically ends the current transaction and begins a new one. Therefore, no SQL statement is ever executed outside of a transaction.

Interrupting Implicit Transactions

ANSI

In ANSI-compliant databases, a transaction is always in effect. BEGIN WORK is not needed, because any COMMIT WORK or ROLLBACK WORK statement that ends a transaction automatically marks the beginning of a new *implicit* transaction. No SQL statement can be executed outside of a transaction.

If an implicit transaction is interrupted by the user, no automatic ROLLBACK WORK occurs. The current transaction is still in progress.

Interrupting Singleton Transactions

A *singleton* transaction occurs for every SQL statement executed outside a transaction. Singleton transactions occur only in databases which are *not* ANSI-compliant.

In a database that is not ANSI-compliant, and that uses transaction logging, the BEGIN WORK statement is *required* to begin a transaction. The database engine treats any SQL statement that you execute outside of a transaction as a *singleton* transaction.

If an interruptable SQL statement (those listed on page 3-236) is within a singleton transaction and is interrupted, the database engine automatically rolls back the current transaction before returning control to the 4GL program. Just as before the SQL statement was interrupted, no transaction is currently in progress.

Interrupting Explicit Transactions

An *explicit* transaction is enclosed between a BEGIN WORK and COMMIT WORK or ROLLBACK WORK statement. Explicit transaction occur only in databases which are *not* ANSI-compliant.

The following table summarizes what the database engine does when an explicit transaction is interrupted:

Database Engine	Database Engine Response to Interrupt
OnLine database engine	All interruptable SQL statements: automatic undo of SQL statement.
SE database engine	All interruptable SQL statements (ALTER INDEX and CREATE INDEX are not interruptable): no automatic undo for current SQL statement (interrupted statement may be in a partially completed state). Current transaction is still in progress.

Handling Interrupted Transactions

When the database engine does *not* perform an automatic rollback, an interrupted transaction can leave the database in an unknown state. In these cases, your program should decide how to proceed.

To check for an interrupted SQL statement, your program can test the values of:

- The **int_flag** built-in variable— if your program contains the DEFER INTERRUPT statement, **int_flag** will have a value of TRUE if the user presses the Interrupt key during an interruptable SQL statement.
- The SQLCA.SQLCODE or **status** built-in variables: if the interruptable SQL statement is preceded by the WHENEVER ERROR CONTINUE statement. This variable will have the value of -213 if the SQL statement failed due to user interruption.

If the database is in a unknown statement, your program should explicitly perform a ROLLBACK WORK statement. The ROLLBACK WORK statement reverses the current transaction while the COMMIT WORK statement commits all modifications made to the database since the beginning of the transaction. To begin a new transaction, you must use the BEGIN WORK statement.

ANSI

In ANSI-compliant databases, the ROLLBACK WORK statement reverses the current implicit transaction and automatically begins a new transaction. No BEGIN WORK statement is needed.

Avoid use of the COMMIT WORK staement when the database is in an unknown state.

The following code fragment checks for an interrupted DELETE statement. This fragment assumes that the database engine is not ANSI-compliant but that it supports transaction logging. Therefore the current transaction is explicit (not a singleton):

```
DEFER INTERRUPT

OPTIONS
 SQL INTERRUPT ON
...
OPEN WINDOW w_purge AT 2,2
  WITH 10 ROWS, 50 COLUMNS
  ATTRIBUTE (BORDER, PROMPT LINE 9)

DISPLAY "ACCOUNT PURGE" AT 1, 2
DISPLAY "Purging customer account of last year's info...."
  AT 3, 2
DISPLAY "Press Cancel to interrupt." AT 4, 2

LET cancelled = FALSE
LET tx_status = 0

BEGIN WORK

  UNLOAD TO filename
    SELECT *
    FROM accthistory
    WHERE customer_num = cust_num
      AND tx_date < start_fiscal

  IF int_flag THEN
    LET int_flag = FALSE
    IF (SQLCA.SQLCODE < 0) THEN
      IF (SQLCA.SQLCODE = -213) THEN
        LET cancelled = TRUE
      ELSE
        LET tx_status = SQLCA.SQLCODE
      END IF
    END IF
  ELSE

    DELETE FROM accthistory
    WHERE customer_num = cust_num
      AND tx_date < start_fiscal

    IF int_flag THEN
      IF (SQLCA.SQLCODE < 0) THEN
```

```
        IF (SQLCA.SQLCODE = -213) THEN
          LET cancelled = TRUE
        ELSE
          LET tx_status = SQLCA.SQLCODE
        END IF
      END IF
    END IF
  END IF

  IF (tx_status < 0) OR cancelled THEN
    ROLLBACK WORK
    IF cancelled THEN
      ERROR "Account purge terminated at user request. ",
            " No information purged."
    ELSE
      ERROR "Account purge terminated (status=",
            tx_status USING "-<<<<<<<<<<<",
            "). No information purged."
    END IF
  ELSE
    COMMIT WORK
    MESSAGE "Account purge complete."
  END IF

CLOSE WINDOW w_purge
```

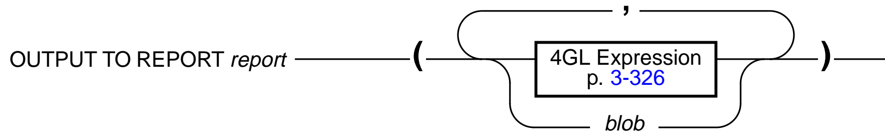
You can refer to the *Informix Guide to SQL: Reference* for more information on data integrity.

References

CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MENU, MESSAGE, OPEN FORM, OPEN WINDOW, PROMPT

OUTPUT TO REPORT

The OUTPUT TO REPORT statement passes a single set of data values (called an “input record”) to a REPORT statement.



blob is the name of a TEXT or BYTE variable to be passed to the report.

report is the name of a 4GL report by which to format the input record. You must also declare this identifier in a REPORT statement, and invoke the report with a previous START REPORT statement.

Usage

The OUTPUT TO REPORT statement passes data to a report, and instructs 4GL to process and format the data as the next input record of the report.

An *input record* is the ordered set of values returned by the expressions that you list between the parentheses. Returned values are passed to the specified report, as part of the input record. The input record can correspond to a retrieved row from the database, or to a 4GL program record, but 4GL does not require this correspondence.

The members of the input record that you specify in the expression list of the OUTPUT TO REPORT statement must correspond to elements of the formal argument list in the REPORT definition (page 3-260) in their number and their position, and must be of compatible data types (page 3-324).

Arguments of the TEXT or BYTE data types are passed by reference, rather than by value; arguments of other data types are passed by value. A report can use the WORDWRAP operator with the PRINT statement (page 6-50) to display TEXT values. A report cannot display BYTE values; the character string `<byte value>` in output from the report indicates a BYTE value.

You typically include the OUTPUT TO REPORT statement within a WHILE, FOR, or FOREACH loop, so that the program passes data to the report one input record at a time. The portion of the 4GL program that includes START REPORT (page 3-271), OUTPUT TO REPORT, and FINISH REPORT (page 3-100) statements that reference the same *report* is sometimes called the “report driver.” (For more information about 4GL reports, see Chapter 6.)

The following program fragment uses a FOREACH loop to pass input records to a report. Each input record consists of four values:

- The **lname** and **company** values from the columns of a database table.
- The literal string constant "San Francisco".
- The DATE value returned by the TODAY operator.

```
START REPORT cust_list
    ...
FOREACH q_curs INTO p_customer.lname, p_customer.company
    OUTPUT TO REPORT cust_list
        (p_customer.lname,
         p_customer.company,
         "San Francisco",
         TODAY)
END FOREACH
```

The following program creates a report, with default formatting, of all the customers in the **customers** table, and sends the resulting output to a file:

```
DATABASE stores
MAIN
    DEFINE p_customer RECORD LIKE customer.*
    DECLARE q_curs CURSOR FOR
        SELECT * FROM customer
    START REPORT cust_list TO "cust_listing"
    FOREACH q_curs INTO p_customer.*
        OUTPUT TO REPORT cust_list(p_customer.*)
    FINISH REPORT cust_list
END MAIN
REPORT cust_list(r_customer)
    DEFINE r_customer RECORD LIKE customer.*
    FORMAT EVERY ROW
END REPORT
```

References

CALL, FINISH REPORT, PAUSE, REPORT, START REPORT

PAUSE

The PAUSE statement suspends the display of output from a 4GL report to the 4GL screen. The PAUSE statement can only appear in the FORMAT section of a REPORT program block and only affects report output sent to the screen.

```
PAUSE ──────┬──────────────────┬──────────────────────┤
            └─── " string " ───┘
```

string is a quoted string.

Usage

The PAUSE statement affects the behavior of the report output in the 4GL screen. It has no effect on the formatted report output:

- If a PAUSE statement appears in the REPORT definition, the report displays a screenful of output and then pauses. The user needs to press RETURN to view the next screenful of output. If a quoted string is specified, its text appears on the 4GL screen.
- In the absence of a PAUSE statement, the report output scrolls down the 4GL screen.

The PAUSE statement has no effect if you include a REPORT TO clause in the OUTPUT section, or a TO clause in the START REPORT statement. For more information about the PAUSE statement, see page 6-41.

References

NEED, PRINT, REPORT, SCROLL, SKIP, START REPORT

PREPARE

Use the PREPARE statement to parse, validate, and generate an execution plan for SQL statements in a **4GL** program at run time.

```
PREPARE ──────── statement id ──── FROM ─┬─ "string" ──┬──┤
                                         └─ variable ──┘
```

statement id is an SQL statement identifier.

string is a string that contains the SQL statement to be prepared.

variable is a **4GL** character variable that contains the text of the SQL statement to be prepared.

Usage

The PREPARE statement permits your program to assemble the text of an SQL statement at run time and make it executable. This *dynamic* form of SQL is accomplished in three steps:

1. A PREPARE statement accepts statement text as input, either as a quoted string or stored within a character variable. Statement text can contain question mark (?) placeholders to represent values that are to be defined when the statement is executed.
2. An EXECUTE or OPEN statement can supply the required input values and execute the prepared statement once or many times.
3. Resources allocated to the prepared statement can be released later using the FREE statement.

The number of prepared objects in a single program is limited only by available memory. This includes both statement identifiers named in PREPARE statements and cursor declarations that incorporate SELECT, EXECUTE PROCEDURE, or INSERT statements. (To avoid exceeding the limit, use a FREE statement to release some statements or cursors.)

For a list of statements that must be prepared, see "Preparing Statements in 4GL" on page 3-247. Also, note that the *INFORMIX-4GL Quick Syntax* marks SQL statements that must be prepared with a 6.0 icon.

For information on using the EXECUTE, OPEN, and FREE statements with a 6.0-level server, see *Informix Guide to SQL: Syntax*.

Statement Identifier

A PREPARE statement sends the statement text to the database server where it is analyzed. If it contains no syntax errors, the text converts to an internal form. This translated statement is saved for later execution in a data structure that the PREPARE statement allocates. The structure has the name specified by the statement identifier (*statement id)* in the PREPARE statement. Subsequent SQL statements can refer to the statement using the *statement id*.

A subsequent FREE *statement id* statement releases the resources allocated to the statement. After you release the *statement id*, you cannot use it with a cursor or with the EXECUTE statement until you prepare the statement again.

By default, the scope of a statement identifier is global in your **4GL** program. This means that a statement identifier prepared in one **4GL** module can be referenced from another module.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or sequence of statements at a time. You can execute a new PREPARE statement with an existing statement identifier if you wish to bind a given statement identifier to different SQL statement text.

Statement Text

The PREPARE statement can take statement text either as a quoted string or as text stored in a variable. The following restrictions apply to the statement text:

- The text can contain only SQL statements. It cannot contain **4GL** statements.

  Comments preceded by two hyphens (--), or enclosed in curly braces ({ }) are standard in SQL and are allowed in the statement text. The comment ends at the end of the line or at the end of the statement.
- The text can contain either a single SQL statement or a sequence of statements separated by semicolons.
- The only identifiers that you can use are names defined in the database, such as names of tables and columns. Therefore, you cannot prepare a SELECT statement that contains an INTO clause because the INTO clause requires a variable.

  Use a question mark (?) as a placeholder to indicate where data is supplied when the statement executes.

The following example shows a PREPARE statement that includes placeholders for values that are to be input:

```
PREPARE new_cust FROM
    "INSERT INTO customer(fname,lname) VALUES(?,?)"
```

Preparing Statements in 4GL

The following section lists statements that must be prepared, may be prepared, and cannot be prepared in a **4GL** program. It also describes how to execute stored procedures from within **4GL**.

Statements that Must Be Prepared

The following SQL statements must be prepared to use them in a **4GL** program. (These statements require preparing because they are features specific to the 5.0 or 6.0 engine. **4GL** supports these statements only if you prepare them.)

ALTER OPTICAL CLUSTER
CREATE OPTICAL CLUSTER
CREATE PROCEDURE
CREATE PROCEDURE FROM
CREATE SCHEMA
CREATE TRIGGER
DROP OPTICAL CLUSTER
DROP PROCEDURE
DROP TRIGGER
EXECUTE PROCEDURE
RELEASE
RESERVE
SET CONSTRAINTS
SET DEBUG FILE TO
SET MOUNTING TIMEOUT
SET OPTIMIZATION

Statements that May Need to Be Prepared

The following SQL statements require you to prepare them only if you are using a 5.0 or 6.0 engine feature in the statement. For example, if you use the PUBLIC or PRIVATE clause of the CREATE SYNONYM statement, you need to prepare the CREATE SYNONYM statement. However, if you do not include the PUBLIC or PRIVATE clause, you do not need to prepare the statement.

ALTER TABLE
CREATE SYNONYM
CREATE TABLE
GRANT
INSERT INTO
REVOKE
UPDATE STATISTICS

To see exactly what part of the syntax requires you to prepare the statement, see the *INFORMIX-4GL Quick Syntax*.

Preparing a SELECT Statement

You can prepare a SELECT statement. If the SELECT statement includes the INTO TEMP clause, you can execute the prepared statement with an EXECUTE statement. If it does not include the INTO TEMP clause, the statement returns an indeterminate number of rows of data. Use DECLARE cursor to establish a cursor and then either the FOREACH statement or the OPEN and FETCH cursor statements to retrieve the rows.

A prepared SELECT statement can include a FOR UPDATE clause. This clause normally is used with the DECLARE statement to create an update cursor. The following example shows a SELECT statement with a FOR UPDATE clause:

```
PREPARE up_sel FROM
    "SELECT * FROM customer ",
    "WHERE customer_num between ? and ? ",
    "FOR UPDATE"

DECLARE up_curs CURSOR FOR up_sel

OPEN up_curs USING low_cust, high_cust
```

Statements that Cannot Be Prepared

You cannot prepare the following statements:

ALLOCATE DESCRIPTOR
CHECK TABLE
CLOSE
CONNECT
DEALLOCATE DESCRIPTOR
DECLARE
DESCRIBE
DISCONNECT
EXECUTE
EXECUTE IMMEDIATE
FETCH
FLUSH
FREE
GET DESCRIPTOR
GET DIAGNOSTICS
INFO
LOAD
PUT
OPEN
OUTPUT
PREPARE
REPAIR TABLE
SET CONNECTION
SET DESCRIPTOR
UNLOAD
WHENEVER

Additionally, you cannot use the following statements in statement text that contains multiple statements separated by semicolons:

CLOSE DATABASE
CREATE DATABASE
DATABASE
DROP DATABASE
SELECT
START DATABASE

Thus, a SELECT statement is not allowed in a multi-statement prepare; the statements that could cause the current database to be closed in the middle of executing the sequence of statements are also not allowed. For general information about multi-statement prepares, see "Preparing Sequences of Multiple SQL Statements" on page 3-252.

Executing Stored Procedures Within a PREPARE Statement

You can include a stored procedure in a **4GL** program by doing the following:

1. Put the text of the CREATE PROCEDURE statement in a file.

   Use Stored Procedure Language statements to define the procedure.
2. Use a PREPARE statement to prepare a CREATE PROCEDURE FROM statement that refers to the text file created in Step 1.
3. Use an EXECUTE statement to execute the prepared statement, which then compiles the stored procedure.

Note: *The Stored Procedure Language is not a part of the* **4GL** *language. You cannot include these statements directly within a* **4GL** *program; doing so causes compile errors.*

You can explicitly invoke stored procedures from within your **4GL** program by preparing and executing the following SQL statements: CREATE PROCEDURE FROM, DROP PROCEDURE, EXECUTE PROCEDURE.

Also, you may implicitly invoke a stored procedure through a reference to that procedure within the context of an SQL expression. For example, the reference to **avg_price( )** in the following SELECT statement implicitly invokes the stored procedure having the name **avg_price**.

```
SELECT
    manu_code, unit_price, (avg_price(1) - unit_price) VARIANCE
    FROM stock
    WHERE stock_num = 1
```

Such implicit references to stored procedures do not require the statement to be prepared since the server processes them in a manner that is transparent to the **4GL** program.

See Chapter 14 of the *Informix Guide to SQL: Tutorial*, Version 6.0 for complete information on creating and executing stored procedures. See the *Informix Guide to SQL: Syntax* for a full description of the CREATE PROCEDURE statement.

Using Parameters in Prepared Statements

You can pass values to a prepared statement when you prepare the statement or at execution time.

Preparing Statements when Parameters Are Known

In some prepared statements, all needed information is known at the time the statement is prepared. Although all parts of the statement are known prior to the prepare, they also can be derived dynamically from program input. In the following example, user input is incorporated into a SELECT statement, which is then prepared and associated with a cursor:

```
DEFINE u_po LIKE orders.po_num
PROMPT "Enter p.o. number please: " FOR u_po
PREPARE sel_po FROM
    "SELECT * FROM orders ",
    "WHERE po_num = '", u_po, "'"
DECLARE get_po CURSOR FOR sel_po
```

Preparing Statements that Receive Parameters at Execution

In some statements, parameters are unknown when the statement is prepared because a different value can be inserted each time the statement is executed. In these statements, you can use a question mark (?) placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following example shows some uses of question mark (?) placeholders:

```
PREPARE s3 FROM
    "SELECT * FROM customer WHERE state MATCHES ?"

PREPARE in1 FROM
    "INSERT INTO manufact VALUES (?,?,?)"

PREPARE update2 FROM
    "UPDATE customer SET zipcode = ?"
    "WHERE CURRENT OF zip_cursor"
```

You can use a placeholder only to supply a value for an expression. You cannot use a question mark (?) placeholder to represent an identifier such as a database name, a table name, or a column name.

The USING clause is available in both OPEN (for statements associated with a cursor) and EXECUTE (all other prepared statements) statements. For example:

```
DEFINE zip LIKE customer.zipcode
PREPARE zip_sel FROM
    "SELECT * FROM customer WHERE zipcode MATCHES ?"
DECLARE zip_curs CURSOR FOR zip_sel
PROMPT "Enter a zipcode: " FOR zip
OPEN zip_curs USING zip
```

If the prepared SELECT statement contains a question mark (?) placeholder, you cannot execute the statement with a FOREACH statement; you must use the OPEN, FETCH, and CLOSE group of statements.

Preparing Statements with SQL Identifiers

You cannot use question mark (?) placeholders for SQL identifiers such as a table name or a column name; you must specify these identifiers in the statement text when you prepare it.

However, if these identifiers are not available when you write the statement, you can construct a statement that receives SQL identifiers from user input. In the following example, the name of the column is supplied by the user and inserted in the statement text before the PREPARE statement. The search value in that column also is taken from user input, but it is supplied to the statement with a USING clause:

```
DEFINE  column_name CHAR(30),
        column_value CHAR(40),
        del_str CHAR(100)

PROMPT "Enter column name: " FOR column_name

LET del_str =
    "DELETE FROM customer WHERE ",
    column_name CLIPPED, " = ?"
PREPARE de4 FROM del_str

PROMPT "Enter search value in column ",column_name, ":"
    FOR column_value

EXECUTE de4 USING column_value
```

Preparing Sequences of Multiple SQL Statements

You can execute several SQL statements as one action if you include them in the same PREPARE statement. Multi-statement text is processed as a unit; actions are not treated sequentially. Therefore, multi-statement text cannot include statements that depend on action that occurs in a previous statement in the text. For example, you cannot create a table and insert values into that table in the same prepared block. Avoid placing BEGIN WORK and COMMIT WORK statements with other statements in a multi-statement prepare.

In most situations, **4GL** returns error status information on the first error in the multistatement text. No indication exists of which statement in the sequence causes an error. You can use **SQLCA** to find the offset of the SQLERRD(5) errors. For complete information about **SQLCA** and error-status information, see "Exception Handling" on page 2-23.

The following example updates the **stores2** database by replacing existing manufacturer codes with new codes. Since the **manu_code** columns are potential join columns that link four of the tables, the new codes must replace the old codes in three tables:

```
DATABASE stores2
MAIN
    DEFINE code_chnge RECORD
        new_code LIKE manufact.manu_code,
        old_code LIKE manufact.manu_code
    END RECORD
    sqlmulti CHAR(250)

PROMPT "Enter new manufacturer code: "
    FOR code_chnge.new_code
PROMPT "Enter old manufacturer code: "
    FOR code_chnge.old_code
LET sqlmulti =
    "UPDATE manufact SET manu_code = ? WHERE manu_code = ?;",
    "UPDATE stock SET menu_code = ? WHERE manu_code = ?;",
    "UPDATE items SET manu_code = ? WHERE manu_code = ?;",
    "UPDATE catalog SET manu_code = ? WHERE manu_code = ?;"

PREPARE exmulti FROM sqlmulti
EXECUTE exmulti USING code_chnge.*, code_chnge.*, code_chnge.*
        code_chnge.*
END MAIN
```

Using Prepared Statements for Efficiency

To increase performance efficiency, you can use the PREPARE statement and an EXECUTE statement in a loop to eliminate overhead caused by redundant parsing and optimizing. For example, an UPDATE statement located within a WHILE loop is parsed each time the loop runs. If you prepare the UPDATE statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution. The following example shows how to prepare statements to improve performance:

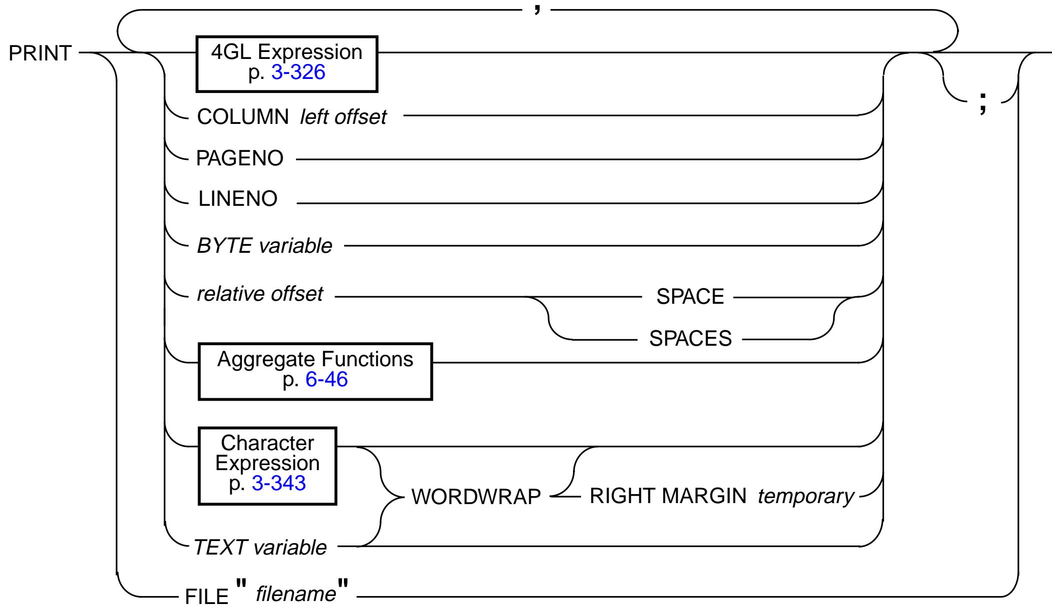
```
PREPARE up1 FROM "UPDATE customer ",
    "SET discount = 0.1 WHERE customer_num = ?"
WHILE TRUE
    PROMPT "Enter Customer Number" FOR dis_cust
    IF dis_cust = 0 THEN
        EXIT WHILE
    END IF
    EXECUTE up1 USING dis_cust
END WHILE
```

References

See the DECLARE, DESCRIBE, EXECUTE, FREE, and OPEN statements in the *Informix Guide to SQL: Syntax*.

PRINT

The PRINT statement produces output from a report. (This statement can appear only in the FORMAT section of a REPORT program block.)



BYTE variable is the identifier of a 4GL variable of data type BYTE.

filename is a character string, enclosed between quotation (") marks, and specifying the name of an ASCII file to include in the output from the report. The *filename* can include a pathname.

left offset is an expression that evaluates to a positive whole number, specifying a character position offset (from the *left margin*) no greater than the difference (*right margin – left margin)*.

relative offset is an expression that evaluates to a positive whole number, specifying an offset (from the *current character position*) no greater than the difference (*right margin – current position)*.

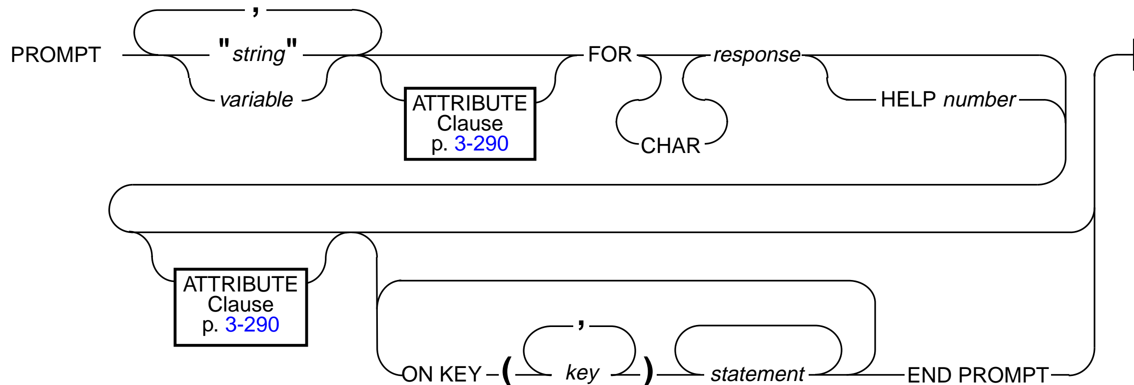
temporary is an expression that evaluates to a positive whole number, specifying the *absolute* position of a *temporary right margin*.

TEXT variable is the identifier of a 4GL variable of data type TEXT.

For details of the syntax and usage of the PRINT statement in 4GL report definitions, see page 6-42.

PROMPT

The PROMPT statement assigns a user-supplied value to a variable.



key is a keyword (see page 3-258) to specify an activation key.

number is a literal integer (page 3-340) to specify a Help message number.

response is the name of a variable to store the response of the user to the PROMPT character string. This cannot be of data type TEXT or BYTE.

statement is a 4GL statement.

string is a quoted string that 4GL displays on the Prompt line.

variable is the name of a CHAR or VARCHAR variable containing a message to the user, typically prompting the user to enter a value.

Usage

4GL takes the following actions when it executes a PROMPT statement:

1. Replaces any variables with their current values.
2. Concatenates the list of values into a single *prompt string*. The total length of this string, plus the user's response, cannot exceed 80 characters.
3. Displays the resulting string on the Prompt line of the current form (or in the Line mode overlay, if it currently covers the 4GL screen).
4. Waits for the user to enter a value.
5. Reads whatever value was entered until the user presses RETURN, and then stores this value in *response variable*.

The prompt string remains visible until the user enters a response.

The following topics are described in this section:

Topic	Page
The PROMPT String	3-256
The Response Variable	3-256
The FOR Clause	3-257
The ATTRIBUTE Clauses	3-257
The HELP Clause	3-258
The ON KEY Blocks	3-258
The END PROMPT Keywords	3-259

The PROMPT String

Depending on whether the Line mode overlay is visible when the PROMPT statement is executed, PROMPT can produce two types of displays:

- If the PROMPT statement is the next interactive statement after a Line mode DISPLAY statement, then the prompt string appears in the Line mode overlay. The prompt string always appears on the bottom line, however, and does not scroll with any subsequent output from Line mode DISPLAY statements. (For more information about using the Line mode overlay, see "Sending Output to the Line Mode Overlay" on page 3-76.)
- If the 4GL screen or any other 4GL window is visible, output appears on the Prompt line of the current 4GL window. If this is not wide enough to display the prompt string, a run-time error occurs.

The default position of the Prompt line is the first line of the current 4GL window. This default position can be changed by either of the following

- A PROMPT LINE specification in the OPEN WINDOW statement.
- A PROMPT LINE specification in the OPTIONS statement.

The Response Variable

The PROMPT statement returns the value entered by the user in the *response variable*. The response variable can be of any data type except TEXT or BYTE. If it is a string, its returned value can include blank spaces. If 4GL cannot convert the value entered by the user to the data type of the response variable, then 4GL cannot assign a value to the response variable; in this case, a negative error code is assigned to the global **status** variable.

The FOR Clause

The FOR clause specifies the name of the response variable to store input from the user. When the user types a response and presses RETURN, 4GL saves the response in the response variable. You can optionally include the CHAR keyword to accept a single character input without requiring that the user press the RETURN key. For example, the following statement checks the prompt input for an upper or lowercase `y`:

```
PROMPT "Do you want to continue: " FOR CHAR ans
IF ans MATCHES "[Yy]" THEN
    CALL next_form()
END IF
```

The ATTRIBUTE Clauses

For general information and syntax of the ATTRIBUTE clause, see page 3-290. This section describes specific information about ATTRIBUTE clauses within a PROMPT statement. You can use the ATTRIBUTE clauses to specify display attributes both for the *prompt value* text and for the *prompt response*.

- The first ATTRIBUTE clause specifies display attributes of the *prompt string* text. The default display attribute for this text is NORMAL.
- The second ATTRIBUTE clause specifies display attributes of the *prompt response*. The default display attribute for the *prompt response* is REVERSE.

Display attributes specified in the PROMPT statement temporarily override any display attributes specified in OPTIONS or OPEN WINDOW statements. The following statement prompts for a delivery date. Here 4GL displays the prompt string in *yellow* on color monitors (or in *bold* on monochrome monitors; see page 3-291). The value that the user enters is displayed in *blue* on monitors that support color, and in *dim* on monochrome monitors.

```
PROMPT "Enter the preferred delivery day for ",
        customer_num, " "
    ATTRIBUTE (YELLOW)
    FOR del_day
    ATTRIBUTE (BLUE)
    ...
END PROMPT
```

The HELP Clause

This clause specifies a literal integer (page 3-340) that returns the number of a Help message for the PROMPT statement. 4GL displays the Help message in the Help window (page 2-22) if the user presses the Help key from the response field. By default, the Help key is CONTROL-W. You can redefine the Help key by using the OPTIONS statement.

You create Help messages in an ASCII file whose filename you specify in the HELP FILE clause of the OPTIONS statement. Use the **mkmessage** utility (as described in Appendix B to create a run-time version of the Help file. Run-time errors occur in these situations:

- 4GL cannot open the Help file.
- You specify a *number* that is not in the Help file.
- You specify a *number* outside the range from -32,767 to 32,767.

The ON KEY Blocks

An ON KEY block executes a series of statements when the user presses one of the specified keys. If the user presses a specified key, control passes to the statements specified in the ON KEY block. After completing the ON KEY block, 4GL passes control to the statements following the END PROMPT statement. In this case, the value of the response variable is undetermined.

You can specify the following in uppercase or lowercase for *key name*:

ACCEPT	HELP	NEXT *or*	RETURN *or* ENTER
DELETE	INSERT	NEXTPAGE	RIGHT
DOWN	INTERRUPT	PREVIOUS *or*	TAB
ESC *or* ESCAPE	LEFT	PREVPAGE	UP

F1 *through* F64

CONTROL-*char* (except A, D, H, I, J, L, M, R, or X)

Here you can substitute NEXTPAGE for NEXT, and PREVPAGE for PREVIOUS.

The following table lists keys that require special consideration before you assign them in an ON KEY clause:

Key	Special Considerations
ESC or ESCAPE	You must use the OPTIONS statement to specify another key as the Accept key because ESCAPE is the default Accept key.
INTERRUPT	You must execute a DEFER INTERRUPT statement. When the user presses the Interrupt key under these conditions, **4GL** executes the ON KEY block statements and sets **int_flag** to nonzero, but does not terminate the PROMPT statement. **4GL** also executes the statements

	in this ON KEY clause if the DEFER QUIT statement has executed and the user presses the Quit key. In this case, **4GL** sets **quit_flag** to non-zero.
CONTROL-*char*	
A, D, H, L, R, and X	**4GL** reserves these keys for field editing.
I, J, and M	The regular meaning of these keys (TAB, LINEFEED, and RETURN, respectively) is lost to the user. Instead, the key is trapped by **4GL** and used to activate the commands in the ON KEY clause. For example, if CONTROL-M appears in an ON KEY clause, the user cannot press RETURN to advance the cursor to the next field. If you must include one of these keys in an ON KEY clause, be careful to restrict the scope of the clause to specific fields.

You may not be able to use other keys that have special meaning to your version of the operating system. For example, CONTROL-C, CONTROL-Q, and CONTROL-S specify the Interrupt, XON, and XOFF signals on many systems.

The next statement specifies two ON KEY clauses. If the user presses CONTROL-B, 4GL calls the **set_day( )** function and sets the **del_day** variable to the value returned by **set_day**. If the user presses F6 or CONTROL-F, the **delivery_help( )** function is invoked:

```
PROMPT "Enter the preferred delivery day for ",
        customer_num, " "
    ATTRIBUTE (YELLOW)
    FOR del_day
    ON KEY (CONTROL_B)
        LET del_day = set_day()
    ON KEY (F6, CONTROL_F)
        CALL delivery_help()
END PROMPT
```

The END PROMPT Keywords

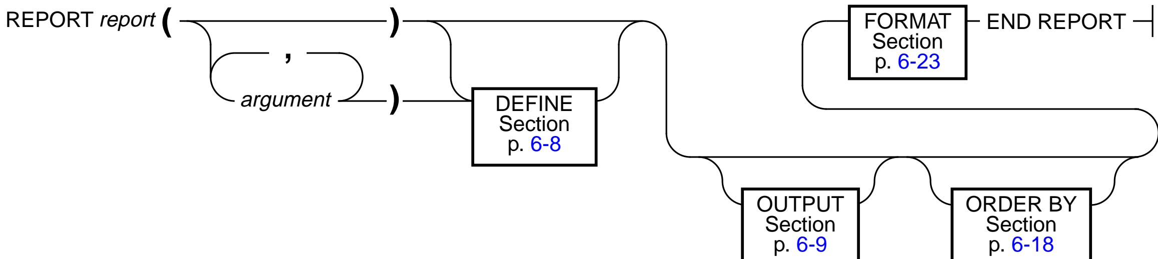
The END PROMPT keywords indicate the end of the PROMPT statement. These keywords are required only if you specify an ON KEY block. Place the END PROMPT keywords after the last statement of the last ON KEY block.

References

DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, OPEN WINDOW, OPTIONS

REPORT

The REPORT statement declares the identifier and defines the format of a 4GL report. (For details on the REPORT statement syntax and usage, see Chapter 6.)



argument is the name of a formal argument (also called a *parameter*) corresponding to a value that the calling routine passes to the report.

report is the name that you assign to the report.

Usage

This statement defines a REPORT program block, just as the FUNCTION statement defines a function. You can execute a report from the MAIN program block or from a function, but the REPORT statement *cannot* appear within the MAIN statement, nor in a FUNCTION definition, nor in another REPORT statement. To create a 4GL report, you must do the following:

1. Use the REPORT statement to describe how to format data in the report.
2. Write a *report driver* that passes data to the report.

The report driver typically uses a loop (such as WHILE or FOREACH) in conjunction with the following 4GL statements to process the report:

- START REPORT (to invoke the REPORT routine)
- OUTPUT TO REPORT (to send data to the REPORT routine for formatting)
- FINISH REPORT (to complete execution of the REPORT routine)

Note: *Unlike a FUNCTION program block, a 4GL REPORT routine is not reentrant. If you execute a START REPORT statement that references a report that is already running, then the report is reinitialized, and output may be unpredictable.*

The Report Prototype

The report name must immediately follow the REPORT keyword. Follow the guidelines for 4GL identifiers (page 2-9) when assigning a name to a report. The name must be unique among function and report names within the 4GL program. Its scope is the entire 4GL program.

The list of formal arguments of the report must be enclosed in parentheses and separated by commas. These are local variables that store values that the calling routine passes to the report. The compiler issues an error unless you declare their data types in the subsequent DEFINE section (page 6-8). You can include a program record in the formal argument list, but you cannot append the **.*** symbols to the name of the record. Arguments can be of any data type except ARRAY, or a record with an ARRAY member.

When you call a report, the formal arguments are assigned values from the argument list of the OUTPUT TO REPORT statement. These *actual arguments* that you pass must match, in number and position, the formal arguments of the REPORT statement. The data types must be compatible (page 3-319), but they need not be identical. 4GL can perform some conversions between compatible data types. The names of the actual arguments and the formal arguments do not have to match.

The Report Program Block

The REPORT definition must include a FORMAT section, and can also include DEFINE, OUTPUT, and ORDER BY sections, as described in Chapter 6. You must declare the data types of the formal arguments and of any local variables in the DEFINE section of the report, which must immediately follow the formal argument list. Within the REPORT program block, these variables take precedence over any global or module variables of the same name. Variables local to the 4GL report cannot be referenced outside of the report, and they do not retain values between invocations of the report. You must include the following in the list of formal arguments:

- All the values for each row sent to the report in the following cases:
  - If you include an ORDER BY section or GROUP PERCENT(*) function.
  - If you use a global aggregate function (one over all rows of the report) anywhere in the report, except in the ON LAST ROW control block.
  - If you specify the FORMAT EVERY ROW default format.
- Any variables referenced in the following group control blocks:
  - AFTER GROUP OF
  - BEFORE GROUP OF

The END REPORT Keywords

The END REPORT keywords terminate the REPORT program block. An example near the end of this section (page 3-262) illustrates a report driver.

The following program fragment briefly illustrates some of the components of the REPORT statement. This example creates a report named **simple** that displays on the screen in default format all the rows from the **customer** table:

```
DECLARE simp_curs CURSOR FOR SELECT * FROM customer
START REPORT simple
FOREACH simp_curs INTO cust.*
    OUTPUT TO REPORT simple(cust.*)
END FOREACH
FINISH REPORT simple
    ...
REPORT simple (x)
    DEFINE x RECORD LIKE customer.*
    FORMAT EVERY ROW
END REPORT
```

Two-Pass Reports

A *two-pass report* is one that creates a temporary table. The REPORT creates a temporary table if it includes any of the following:

- An ORDER BY section without the EXTERNAL keyword.
- The GROUP PERCENT(*) aggregate function anywhere in the report.
- Any aggregate function outside the AFTER GROUP OF control block.

The FINISH REPORT statement uses values from these tables to calculate any global aggregates, and then deletes the tables.

A two-pass report requires that the 4GL program be connected to a database when the report runs. See the DATABASE statement for information on how to specify a current database at run time (page 3-60).

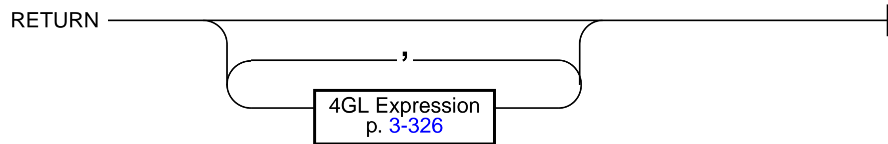
Note: *If the DEFINE section uses the LIKE keyword to declare local variables of the report indirectly, you must also include a DATABASE statement in the same module as the REPORT statement, but before the first program block, to specify a default database at compile time (page 3-59).*

References

DATABASE, DEFINE, FINISH REPORT, OUTPUT TO REPORT, START REPORT

RETURN

The RETURN statement transfers control of execution from a FUNCTION program block (page 3-111). It can also return values to the calling routine. (This statement can appear only within a FUNCTION program block.)



Usage

The RETURN statement can occur only in the definition of a function. This statement tells 4GL to exit from the function, and to return program control to the calling routine. (The *calling routine* is the MAIN, FUNCTION, or REPORT program block that contains the statement that invoked the function.)

You can use the RETURN statement in either of two ways:

- Without values, to control the flow of program execution.
- With a list of one or more values, to control the flow of program execution, and to return values to the calling statement.

If 4GL does not encounter a RETURN statement, it exits from the function after encountering the END FUNCTION keywords.

The List of Returned Values

You can specify a list of one or more expressions as values to return to the calling routine. You can use the *record.** or the THRU or THROUGH notation to specify all or part of a list of the member variables of a record.

If the RETURN statement specifies one or more values, you can do either of the following to invoke the function:

- Explicitly execute a CALL statement with a RETURNING clause.
- Invoke the function implicitly within an expression (in the same way that you would specify a variable or a list of variables).

If the function does not return any values, you must use the CALL statement (without the RETURNING clause) to invoke the function.

The Data Types of Returned Values

4GL compares the list of expressions in the RETURN statement to arguments in the RETURNING clause of the CALL statement. A compile-time error is issued if any of these arguments do not agree with the RETURN expression list in number or position, or if data types are incompatible (page 3-324).

Similarly, if the function is invoked implicitly in an expression (page 3-332), the RETURN statement is checked for agreement with the number and data types of the values that are required by the context of the calling statement.

You cannot return variables of the ARRAY data type, nor RECORD variables that contain ARRAY members. You can, however, return records that do not include ARRAY members. This example returns the values of **whole_price** and **ret_price** to the CALL statement. 4GL then assigns the **whole_price** and **ret_price** variables to the **wholesale** and **retail** variables in the **price** record.

```
MAIN
    DEFINE price RECORD wholesale, retail MONEY
        END RECORD
    ...
    CALL get_cust() RETURNING price.*
    ...
END MAIN
FUNCTION get_cust()
    DEFINE whole_price, ret_price MONEY
    ...
    RETURN whole_price, ret_price
END FUNCTION
```

You cannot specify variables of the BYTE or TEXT data types in the RETURN statement, just as you cannot include those data types in the RETURNING clause of a CALL statement. Since 4GL passes variables of large data types by reference, any changes made to a BYTE or TEXT variable within a function becomes visible within the calling routine without being returned.

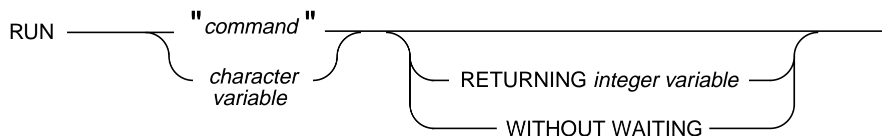
4GL allocates 5 kilobytes of memory to store character strings returned by functions, in 10 blocks of 512 bytes. A returned character value can be no larger than 511 bytes (because every string requires a terminating ASCII 0), and no more than 10 of these 511-byte strings can be returned. You can use TEXT variables to pass longer character values by reference (as described on page 3-18), rather than using the RETURN statement.

References

CALL, EXIT PROGRAM, FUNCTION, WHENEVER

RUN

The RUN statement executes an operating system command line.



character variable	is the name of a CHAR or VARCHAR variable containing a command line for the operating system to execute.
command	is a quoted string (page 3-343) that specifies a command line for the operating system to execute.
integer variable	is the name of an INT or SMALLINT variable.

Usage

The RUN statement executes an operating system command line. You can even run a second 4GL application as a secondary process, provided that only one of the 4GL applications accesses a database. When the command terminates, 4GL resumes execution.

For example, the following statement executes the command line specified by the element **i** of the array variable **charval**, where **i** is an INT or SMALLINT variable:

```
RUN charval[i]
```

Unless you specify WITHOUT WAITING, RUN also has these effects:

1. Causes execution of the current 4GL program to pause.
2. Displays any output from the specified *command* in a new 4GL window.
3. After that command completes execution, 4GL closes the new 4GL window, and restores the previous display in the 4GL screen.

If you specify WITHOUT WAITING (page 3-267), all of these effects except the last are suppressed, so that the command line typically executes without any effect on the visual display.

You can specify the name of a character variable that contains the command line, or the command can follow the RUN keyword as a quoted string.

The RETURNING Clause

The RETURNING clause saves the termination status code of a command in a 4GL variable. You can then examine this variable in your program to determine the next action to take. A status code of zero usually indicates that the command terminated normally. Non-zero exit status codes usually indicate that an error or a signal caused execution to terminate.

You can only use this clause if RUN invokes a 4GL program that contains an EXIT PROGRAM statement. When the program that RUN specifies completes execution, the *integer variable* contains two bytes of termination status information:

- The low byte contains the termination status of whatever RUN executes. You can recover this by calculating the value of (*integer value* modulo 256).
- The high byte contains the low byte from the EXIT PROGRAM statement of the 4GL program that RUN executes. You can recover this returned code by dividing *integer value* by 256.

For example, suppose that a program consisted of these 4GL statements:

```
MAIN
  DEFINE ret_int INT
  LET ret_int = 5
  EXIT PROGRAM (ret_int)
END MAIN
```

The following program fragment uses RUN to invoke the compiled version of the previous program, whose filename is stored in variable **prog1**:

```
DEFINE expg_code, stat_code, ret_int INT,
       prog1 CHAR(20)
    . . .
    RUN prog1 RETURNING ret_int
    LET stat_code = (ret_int MOD 256)
    IF stat_code <> 0 THEN
       MESSAGE "Unable to run the ", prog1, " program."
    END IF
    LET expg_code = (ret_int/256)
    DISPLAY " Code from the ", prog1, " program is ", expg_code
```

Unless an error or signal terminates the program before the EXIT PROGRAM statements is encountered, the displayed value of **expg_code** is 5. You should exercise caution in interpreting the integer variable, however, because under some circumstances the quotient (*variable*)/256 may not be the actual status code value that the command line returned.

If an Interrupt signal terminates the program, the integer value is 256.
If a Quit signal causes the termination, the integer value is (3*256), or 758.

If a 4GL program that RUN executes can be terminated by various actions of the user, you could include several EXIT PROGRAM (*number*) statements with different *number* values in different parts of the program. Examination of the code returned by RUN could indicate which EXIT PROGRAM statement (if any) was encountered during execution.

The WITHOUT WAITING Clause

The WITHOUT WAITING clause lets you execute a secondary application in the background. The syntax of WITHOUT WAITING is illustrated in the following example:

```
RUN "/$INFORMIXDIR/bin/fglgo /home/elke/sub.4gi" WITHOUT WAITING
```

Each 4GL application must have its own MAIN routine. The two programs cannot share variable scope. Each must be independently terminated, either by executing an END MAIN or EXIT PROGRAM statement in 4GL.

The WITHOUT WAITING clause is useful if you know that the command will take some time to execute, and your **4GL** program does not need the result to continue. Because RUN WITHOUT WAITING executes the specified command line as a *background process*, it generally does not affect the visual display.

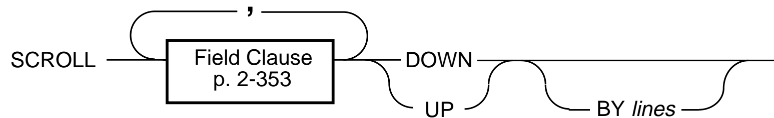
A common way to use RUN WITHOUT WAITING is to generate reports in the background.

References

CALL, FUNCTION, START REPORT

SCROLL

The SCROLL statement specifies vertical movement of displayed values in all or some of the fields of a screen array within the current form.



lines is a literal integer (page 3-340), or the name of a variable containing an integer value, that specifies how far (in lines) to scroll the display.

Usage

Here 1 ≤ *lines* ≤ *size*, for *size* the number of lines in the screen array, and *lines* the positive whole number specified in the BY clause, indicating how many lines to move the displayed values vertically in the specified fields of a screen array. If you omit the BY *lines* specification, the default value is one (1) line.

Specify UP to scroll the data towards the top of the form, or DOWN to scroll toward the bottom of the form. For example, the following statement moves *up* by one line all the displayed values in the **sc_item** screen array, and fills with blanks all the fields of the last (that is, the bottom) screen record:

```
SCROLL sc_item.* UP
```

The BY clause indicates how many lines upwards or downwards to move the data; if you omit it, as in the previous example, the default is 1 line in the specified direction. This example moves values in two fields *down* by 3 lines:

```
SCROLL stock_num, manu_code DOWN BY 3
```

The SCROLL statement ignores any bracket notation (like **sc_item[3].***) that references a single record within the array; 4GL always scrolls values in the specified fields of *every* screen record.

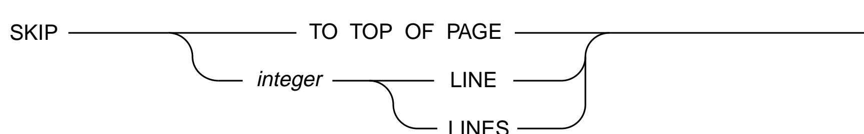
If you use the SCROLL statement, it is your responsibility to keep track of the data left on the screen. Many developers prefer to have the user use the scrolling keys of the INPUT ARRAY (page 3-172) or DISPLAY ARRAY (page 3-91) statements, rather than SCROLL, to scroll through screen records.

References

DISPLAY ARRAY, INPUT ARRAY

SKIP

The SKIP statement inserts blank lines into a report, or finishes the current page. (It can appear only in the FORMAT section of a REPORT program block.)



lines is a literal integer (page 3-340), specifying how many blank lines to insert below the current line.

Usage

The SKIP statement inserts blank lines into REPORT output, or advances the current print position to the top of the next page; see also Chapter 6. The LINE and LINES keywords are synonyms in the SKIP statement (and only here).

Output from any *page trailer* or *page header* block appears in its usual location. The following program fragment produces a list of names and addresses:

```
FIRST PAGE HEADER
   PRINT COLUMN 30, "CUSTOMER LIST"
   SKIP 2 LINES
   PRINT "Listings for the State of ", thisstate
   SKIP 2 LINES
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
PAGE HEADER
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
ON EVERY ROW
   PRINT customer_num USING "####",
   COLUMN 12, fname CLIPPED, 1 SPACE,
      lname CLIPPED, COLUMN 35, city CLIPPED, ", " , state,
      COLUMN 57, zipcode, COLUMN 65, phone
```

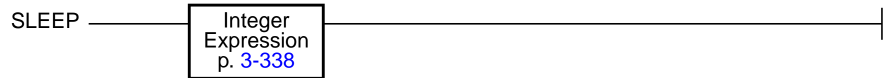
The SKIP LINES statement cannot appear within a CASE statement, a FOR loop, nor a WHILE loop. The SKIP TO TOP OF PAGE statement cannot appear in a FIRST PAGE HEADER, PAGE HEADER, nor PAGE TRAILER control block.

References

NEED, OUTPUT TO REPORT, PAUSE, PRINT, REPORT, START REPORT

SLEEP

The SLEEP statement suspends execution of the 4GL program for a specified positive whole number of seconds.



Usage

The SLEEP statement causes the program to pause for the specified number of seconds. This can be useful, for example, when you want the screen display to remain visible long enough for the user to read it. The following statement displays a screen message, and then waits 3 seconds before erasing it:

```
MESSAGE "Row has been added."
SLEEP 3
MESSAGE " "
```

In contexts where the PROMPT statement is valid, an alternative to SLEEP is the PROMPT statement. The following example suspends program execution until the user acknowledges a screen message by providing keyboard input:

```
PROMPT "Row was added. Press RETURN to continue:" FOR reply
```

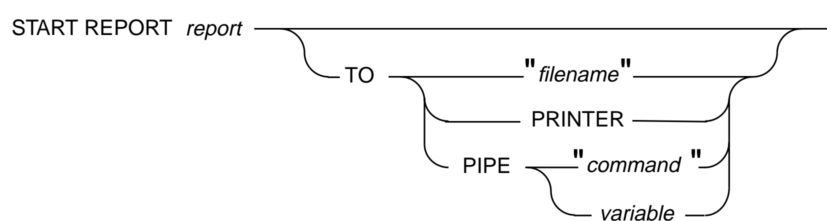
Here the screen display remains visible until the user presses RETURN (or enters anything), rather than for a fixed time interval. Entered keystroke(s) are stored in the **reply** variable, but their actual value can be ignored.

References

DISPLAY, EXPRESSION, MESSAGE, PROMPT

START REPORT

The START REPORT statement begins processing a 4GL report. (For a discussion of 4GL reports, see Chapter 6.)



command is a quoted string, containing a command to receive report output.

filename is a quoted string, containing the name of the file to receive the report output. The *filename* can include a pathname.

report is the name of a report, as declared in a REPORT statement.

variable is a character variable, containing a command to receive output.

Usage

Use the START REPORT statement in a report driver to begin processing a report. This statement does the following:

- Identifies a REPORT routine to format the input records.
- Specifies where to send the formatted output from the REPORT routine.
- Initializes any page headers in the FORMAT section of the report.

The START REPORT statement typically occurs just before a FOR, FOREACH, or WHILE loop in which you use the OUTPUT TO REPORT statement to send input records to the report. After the loop has terminated, you can execute the FINISH REPORT statement to complete the report.

Do not use the START REPORT statement to reference a *report* that is already running; if you do, any output will be unpredictable but probably useless.

The TO Clause

The optional TO clause determines where to direct the output from the report. If you specify a TO clause, 4GL ignores any REPORT TO clause (page 6-13) in the OUTPUT section of the REPORT definition.

If the OUTPUT TO REPORT statement sends an empty set of data records to the report, the report produces no output, and the TO clause has no effect, even if headers, footers, and other formatting control blocks are specified. The TO clause can send the output to any of three destinations:

- To a printer
- To a file
- To another program, command line, or shell script

The sections that follow describe these three options of the TO clause.

If you omit the TO clause of START REPORT, 4GL sends the report output to the destination that you indicate in the optional REPORT TO specification of the OUTPUT section of the REPORT definition, as described on page 6-13. If neither START REPORT nor the REPORT definition specifies a destination, then output is sent by default to the Report window (page 6-14).

The TO PRINTER Option

If you use the TO PRINTER option, 4GL sends the report output to the device or program specified by the DBPRINT environment variable. If you do not set this variable, 4GL sends output to the default printer of the system. For information about setting DBPRINT and other environment variables that 4GL uses, see Appendix D.

The following statement sends output from report **cust_list** to the printer:

```
START REPORT cust_list TO PRINTER
```

If you want to send the output to a printer other than the default printer, you can do one of the following:

- Set the DBPRINT environment variable.
- Use the TO *filename* option to send the output to a file, and then send the file to a printer.
- Use the TO PIPE option to direct the output to a file, and then send the file to a printer.

The TO *filename* Option

If you use the TO *filename* option, 4GL sends the report output to the file specified. The *filename* specification must be a quoted string, not a program variable. For example, the next statement sends output from the **cu_list** report to the file **outfile**:

```
START REPORT cu_list TO "outfile"
```

The next program creates a report, with default formatting, describing all the customers in the **customers** table, and saves it in the **cust_lst** file.

```
DATABASE stores2

MAIN
    DEFINE p_customer RECORD LIKE customer.*
    DECLARE q_curs CURSOR FOR
        SELECT * FROM customer
    START REPORT cust_list TO "cust_lst"
    FOREACH q_curs INTO p_customer.*
        OUTPUT TO REPORT cust_list(p_customer.*)
    END FOREACH
    FINISH REPORT cu _list
END MAIN

REPORT cust_lst(r_customer)
    DEFINE r_customer RECORD LIKE customer.*
    OUTPUT REPORT TO PRINTER
    FORMAT EVERY ROW
END REPORT
```

Here 4GL ignores the OUTPUT REPORT TO PRINTER specification in the REPORT routine, because the TO *filename* clause of the START REPORT statement overrides any default destination in the REPORT routine. But if the same **cust_list** report were referenced in another START REPORT statement that had no TO clause, then its output would go to the default printer.

The TO PIPE Option

The TO PIPE option sends the report output to a UNIX program, shell script, or command line. You can include command-line arguments in the character string or variable specified for the TO PIPE option. For example, the following statement pipes output from the report to the **more** program:

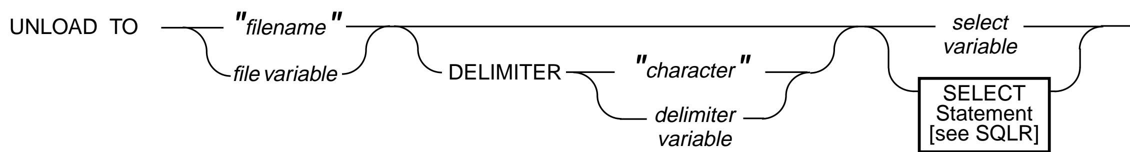
```
START REPORT cust_list TO PIPE "/usr/ucb/more"
```

References

FINISH REPORT, OUTPUT TO REPORT, REPORT

UNLOAD

The UNLOAD statement copies data from the current database to a file.



character is a literal delimiter symbol, enclosed between quotation marks.

delimiter variable is a CHAR or VARCHAR variable containing single symbol to separate adjacent columns within the ASCII representation each row from the database in the output file.

filename specifies the name of an output file in which to store the rows retrieved by the SELECT statement. This can include a pathname.

file variable is a CHAR or VARCHAR variable containing a *filename*.

select variable is a CHAR or VARCHAR variable containing a SELECT statement. (See the *Informix Guide to SQL: Reference* for the syntax of SELECT.)

Usage

The UNLOAD statement must include a SELECT statement (directly, or in a variable) to specify what rows to copy into *filename*. UNLOAD does not delete the copied data. The user must have Select privileges on every column specified in the SELECT statement. (For database-level and table-level privileges, see the GRANT statement in the *Informix Guide to SQL: Reference*.)

The DATABASE statement must first open the database that SELECT accesses.

You cannot use the PREPARE statement to preprocess an UNLOAD statement.

The Output File

The *filename* or *file variable* identifies an output file to store the rows retrieved from the database by the SELECT statement.

A set of values in output representing a row from the database is called an *output record*. A Newline character (ASCII 10) terminates each output record.

The UNLOAD statement represents each value in the output file as a string of ASCII characters, according to the declared data type of the database column:

Data Type	Output Format
character	Trailing blanks are dropped from CHAR and TEXT (but *not* from VARCHAR) values. Backslash (\ ) is inserted before any literal backslash or *delimiter* character, and before Newline anywhere in a VARCHAR value or as the last character in a TEXT value.
number	Values are written as literals (page 3-341) with no leading blanks. MONEY values are represented with no leading currency symbol. Zero values are represented as `0` for INTEGER or SMALLINT columns, and as `0.00` for FLOAT, SMALLFLOAT, DECIMAL, or MONEY columns. SERIAL values are represented as literal integers (page 3-338).
DATE	Values are written in the format *month***/** *day***/** *year* (page 3-349) unless some other format is specified by the DBDATE environment variable.
DATETIME, INTERVAL	Values are written in numeric format *year***-***month***-***day* *hour***:** *minute***:** *second***.** *fraction* as on pages *3-351* and 3-355), including only the time unit values and delimiters. Time units outside the declared precision of the database column are omitted.
BYTE	Values are written in ASCII hexadecimal form, without any added blank or Newline characters. The logical record length of an output file that contains BYTE values can be very long, and thus may be very difficult to print or to edit.

NULL values of any data type are represented by consecutive delimiters in the output file, without any characters between the delimiter symbols.

Quotation (`"`) marks are required around a literal *filename*. The following statement copies any rows where the value of `customer.customer_num` is greater than or equal to 138, and stores them in a file called **cust_file**:

```
UNLOAD TO "cust_file" DELIMITER "!"
    SELECT * FROM customer WHERE customer_num> = 138
```

This produces this output file, **cust_file**:

```
138!Jeffery!Padgett!Wheel Thrills!3450 El Camino!Suite 10!Palo Alto!CA!94306!!
139!Linda!Lane!Palo Alto Bicycles!2344 University!!Palo Alto!CA!94301!(415)323-5400
```

NLS

The UNLOAD statement uses the environment variables DBFORMAT, DBMONEY, LC_NUMERIC, LC_MONETARY, and DBDATE to determine the format of the output file. The precedence of these format specifications is consistent with that of other output facilities (that is, forms and reports). For complete information, see Appendix E.

The DELIMITER Clause

The DELIMITER clause specifies the delimiting character that separates the data contained in each column in a row in the output file. Enclosing quotation (") marks are required around a literal *delimiter* symbol.

The following statement specifies semicolon (;) as the delimiter character:

```
UNLOAD TO "cust.out" DELIMITER ";"
    SELECT fname, lname, company, city
        FROM customer
```

If you omit this clause, then the default delimiter symbol is the value of the DBDELIMITER environment variable, or the vertical bar (| = ASCII 124) if DBDELIMITER is not set. For details of how to set the DBDELIMITER variable, see Appendix D.

Do not specify any of the following characters as the delimiter symbol:

- Hexadecimal numbers (0 through 9, a through f, or A through F)
- Newline or Control-J
- The backslash (\ ) symbol

The Backslash Escape Character

The backslash serves as an escape character in the output file to indicate that the next character is a literal character in a data value. The UNLOAD statement automatically inserts a preceding backslash to prevent literal characters from being interpreted as special characters in the following contexts:

- The backslash symbol anywhere in a CHAR, VARCHAR, or TEXT column.
- The *delimiter* symbol anywhere in a CHAR, VARCHAR, or TEXT column.

- The Newline symbol anywhere in a VARCHAR column, or as the last character in a TEXT column.

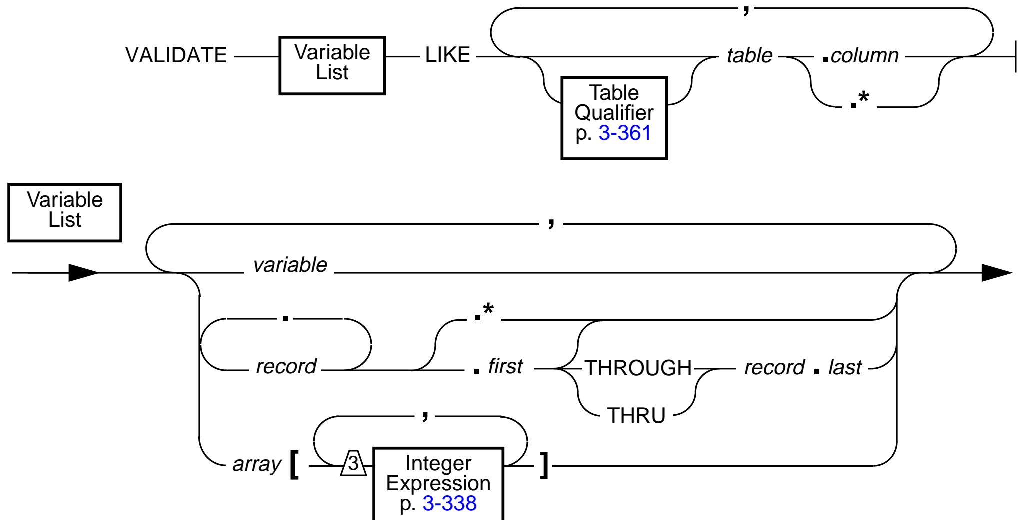
If a LOAD statement (or a TBLOAD command) inserts output from UNLOAD into the database, all escapist backslash symbols are automatically stripped.

References

DATABASE, LOAD

VALIDATE

The VALIDATE statement tests whether the value of a variable is within the range of values for a corresponding column in the **syscolval** table.



array	is the name of a variable of the ARRAY OF INTEGER or ARRAY OF SMALLINT data type that is to be validated.
column	is the name of a column of *table* for which an INCLUDE value exists in the **syscolval** table of the default database.
first	is the name of a member variable of *record* to be validated.
last	is another member that was declared later than *first* in *record.*
record	is the name of a program record to be validated.
table	is the name or synonym of the table or view that contains *column.*
variable	is the name of a variable (of a simple data type) to be validated.

Usage

If your program inserts data from a screen form, then 4GL automatically checks for validation criteria. If your program inserts data into the database from sources other than a screen form, you can apply validation criteria from the **syscolval** table (page 5-69) by using the VALIDATE statement.

You must include a DATABASE statement before the first program block in the same module to specify a default database (page 3-59) at compile time.

This statement has no effect unless the **upscol** utility has assigned INCLUDE values in the **syscolval** table (page 3-280) for at least one of the database columns in the column list (page 3-279) of the VALIDATE statement.

If the value of a variable does not conform with the INCLUDE value in the **syscolval** table (page 3-280), then 4GL sets the **status** variable to -4504. If you specify a list of variables and receive a negative **status** value, you must test the variables individually to detect the non-conforming value.

Because INCLUDE values can be specified only for database columns of simple data types, the list of variables cannot include BYTE nor TEXT variables. You can, however, include members of RECORD variables, or elements of ARRAY variables, if these members or elements are of simple data types.

The LIKE Clause

The LIKE clause specifies the database columns with which to validate the variables.

The variables must match the specified columns in order and number, and must be of the same or compatible data types (page 3-324). You must prefix the name of each *column* with that of the *table*. For example, the following statement validates two variables against two columns in the **stock** table:

```
VALIDATE var1, var2 LIKE stock.stock_num, stock.manu_code
```

In an ANSI-compliant database, you must qualify each table name with that of the *owner* of the table (*owner.table*). The only exception is that you can omit the *owner* prefix for any tables that you own. For example, if you own **tab1**, Krystl owns **tab2**, and Nick owns **tab3**, then you could use this statement

```
VALIDATE var1, var2, var3
    LIKE tab1.var1, krystl.tab2.var2, nick.tab3.var3
```

You can include the owner name in a database that is not ANSI-compliant. If the *owner* is incorrect, 4GL generates an error. For more information, see the *Informix Guide to SQL: Reference.*

You can also reference columns in tables outside the default database. See the section “Table Qualifiers” on page 3-361 for more information. Even if you specify the name of a database in the table qualifier, however, you must also include a DATABASE statement before the first program block in the same module to specify a default database (page page 3-59) at compile time.

The syscolval Table

The VALIDATE statement looks up validation criteria in the INCLUDE column of the **syscolval** table. To enter values into this table, use the **upscol** utility, as described in Appendix B. If a column does not have any INCLUDE value in **syscolval**, then 4GL takes no action. If the current database is not ANSI-compliant, **upscol** creates a single **syscolval** table for all users.

ANSI

In an ANSI-compliant database, each user of the **upscol** utility creates an *owner*.**syscolval** table, which stores validation criteria only for the tables owned by that user. If you omit the *owner* qualifier for a table that you own, your **syscolval** table becomes the source for validation criteria when you compile the program. If the *owner*.**syscolval** table does not exist, the VALIDATE statement takes no action.

OL

The **upscol** utility cannot specify validation criteria for TEXT or BYTE columns. Because of this restriction, the VALIDATE statement cannot reference variables of these large binary data types.

The compiler looks in the default database for **syscolval**. Any changes to **syscolval** after compilation have no effect on the 4GL program, unless you recompile the program.

This example assumes that the **state** field in the **customer** table has validation criteria in **syscolval** that limit the valid states to those in the Western region:

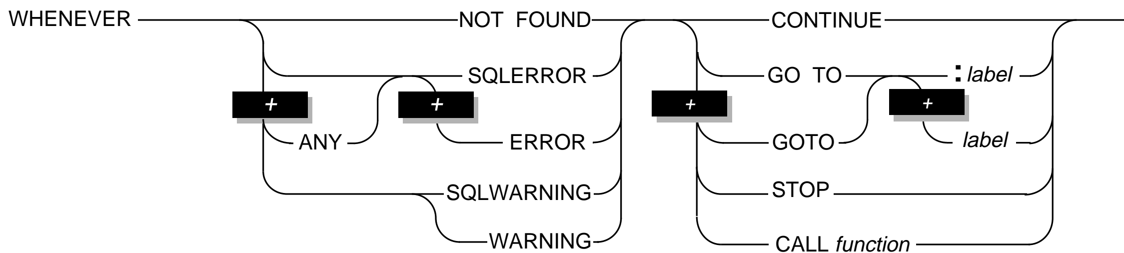
```
INPUT BY NAME p_customer.*
    ...
    AFTER FIELD zipcode
        CALL check_zip(p_customer.zipcode)
        RETURNING state_zip
        WHENEVER ERROR CONTINUE
        VALIDATE state_zip LIKE customer.state
        WHENEVER ERROR STOP
        IF (status < 0) THEN
            ERROR "This zipcode is not in the Western region."
        END IF
    ...
END INPUT
```

References

DATABASE, DEFINE, INITIALIZE, INPUT, INPUT ARRAY, WHENEVER

WHENEVER

The WHENEVER statement traps SQL and 4GL errors, warnings, and end-of-data conditions that may occur during program execution.



function is a function name (without parentheses or argument list) to be invoked if the specified exceptional condition occurs.

label is a statement label (in the same program block) to which 4GL transfers program control when the specified exceptional condition occurs. (Statement labels must be declared with the LABEL statement, as described on page 3-177.)

Usage

This statement can appear only within a MAIN, REPORT, or FUNCTION program block. It can trap errors, warnings, and the NOT FOUND condition at run time. The WHENEVER statement must include two items of information:

- Some type of *exceptional condition*.
- An *action* to take if the specified exceptional condition is detected.

These specifications correspond respectively to the left-hand (*conditions*) and right-hand (*actions*) portions of the syntax diagram that appears on this page.

Using WHENEVER is equivalent to including code after every SQL statement, and (optionally) after certain other 4GL statements to take the specified action if the exceptional condition is detected. Without WHENEVER, program execution immediately stops when a run-time error occurs, unless the database is ANSI-compliant.

If you use WHENEVER ERROR with any option but STOP or CONTINUE, 4GL tests for errors by polling the global variable **status**.

Topics that are discussed in this section include the following:

Topic	Page
The Scope of the WHENEVER Statement	3-282
Exceptional Conditions (ERROR, ANY, NOT FOUND, WARNING)	3-282
Action Options (GO TO, CALL, CONTINUE, STOP)	3-284

The Scope of the WHENEVER Statement

The scope of a WHENEVER statement is from its location in a program block until the next WHENEVER statement with the same exceptional condition in the same module (except that both ERROR and ANY ERROR reset both ERROR and ANY ERROR). Otherwise, the WHENEVER statement remains in effect for that exceptional condition until the end of the module.

For example, the following program has three WHENEVER statements, two of which describe WHENEVER ERROR conditions. In line 4, CONTINUE is specified as the action to take; line 8 specifies STOP as the action for the same ERROR condition. Any errors that 4GL encounters after line 4 but before line 8 are ignored. After line 8, and for the rest of the program, any errors that are encountered cause the program to terminate.

```
MAIN                                                    --1
    DEFINE char_num INTEGER                             --2
    DATABASE test                                       --3
    WHENEVER ERROR CONTINUE                             --4
    PRINT "Program will now attempt first insert."      --5
    INSERT INTO test_color VALUES ("green")             --6
    WHENEVER NOT FOUND CONTINUE                         --7
    WHENEVER ERROR STOP                                 --8
    PRINT "Program will now attempt second insert."     --9
    INSERT INTO test_color VALUES ("blue")              --10
    CLOSE DATABASE                                      --11
    PRINT "Program over."                               --12
END MAIN                                                --13
```

The ERROR Condition

The ERROR keyword directs 4GL to take the specified action if **sqlcode** in the SQLCA global record is negative after any SQL statement, or if a VALIDATE (page 3-278) or screen interaction statement (page 3-228) fails. For example, this statement causes SQL errors to be ignored:

```
WHENEVER ERROR CONTINUE
```

If you do not use any WHENEVER ERROR statements in a program and if, at compile time, the database accessed by the program is not ANSI-compliant, then the default for WHENEVER ERROR is STOP.

ANSI

If you do not use any WHENEVER ERROR statements in a program and if, at compile time, the database accessed by the program is ANSI-compliant, then the default for WHENEVER ERROR is CONTINUE.

Besides checking for errors after SQL statements, the WHENEVER ERROR statement also checks for errors after screen interaction statements (page 3-228) and after VALIDATE statements. (In a WHENEVER statement, and only in this context, SQLERROR is a synonym for ERROR. You cannot, for example, substitute SQLERROR for ERROR in an OPTIONS or ERROR statement.)

Certain errors cannot be trapped by the WHENEVER ERROR statement. Some errors always terminate the program, and others cause 4GL to print an error message and exit prior to the action specified by WHENEVER. For a list of untrappable run-time errors, see "Exception Handling" on page 2-23.

The ANY ERROR Condition

Without ANY, WHENEVER ERROR resets **status** to the **sqlcode** value only if the error occurs during an SQL, VALIDATE, or screen interaction statement. The ANY keyword before ERROR, however, resets **status** after evaluating any 4GL expression. The **-anyerr** command-line option is described in Chapter 1. This can override WHENEVER statements in determining whether the **status** variable is reset when 4GL expressions are evaluated.

The NOT FOUND Condition

If you use the NOT FOUND keywords, SELECT and FETCH statements (and implicit FETCH or SELECT statements in FOREACH or UNLOAD statements) are treated differently from other SQL statements. The NOT FOUND keywords check for the end-of-data condition in the following cases:

- A FETCH attempts to get a row beyond the first or last row in the active set.
- A SELECT statement returns no rows.

In both cases, the **sqlcode** variable is set to 100. The following statement calls the **no_rows( )** function whenever the NOT FOUND condition is detected:

```
WHENEVER NOT FOUND CALL no_rows
```

Note:** Although both NOT FOUND and NOTFOUND indicate the same condition, they cannot be used interchangeably. Use NOTFOUND (one word) in* **status, and use NOT FOUND (two words) in the WHENEVER statement.*

The WARNING Condition

If you use the WARNING keyword (or its synonym SQLWARNING), any SQL statement that generates a warning also produces the action indicated by the WHENEVER WARNING statement. If a warning occurs, the first field of the SQLAWARN record is set to `W`. For example, the following statement causes a program to halt execution whenever a warning condition exists:

```
WHENEVER WARNING STOP
```

The GOTO Option

Use the GOTO clause to transfer control to the statement identified by the specified statement label. The keywords GO TO are a synonym for GOTO.

The label that follows the GOTO keyword must be declared by a LABEL statement in the same FUNCTION, REPORT, or MAIN program block as the current WHENEVER statement. For example, the WHENEVER statement in this program fragment transfers control to the statement labeled **missing:** whenever the NOT FOUND condition occurs:

```
FUNCTION query_data()
    ...
    FETCH FIRST a_curs INTO p_customer.*
    WHENEVER NOT FOUND GO TO :missing
    ...
    LABEL missing:
        MESSAGE "No customers found."
        SLEEP 3
        MESSAGE ""
END FUNCTION
```

If your source module contains more than one program block, you may need to redefine the error condition. For example, suppose that the module contains three functions, and the first function includes a WHENEVER ... GOTO statement and a corresponding LABEL statement. When compilation moves from the first FUNCTION definition to the next, the WHENEVER specification still specifies a jump to the label, but that label is no longer defined in the second FUNCTION block. If the compiler processes an SQL statement within that block before you redefine the action to take for the same condition (for example, to WHENEVER ERROR CONTINUE), then a compilation error results.

To avoid this error, you can reset the error condition by issuing another WHENEVER statement. Alternatively, you can use the LABEL statement to define the same statement label in each function, or you can use the CALL option of WHENEVER to invoke a separate function.

The CALL Option

The CALL clause transfers program control to the specified function. Do not include parentheses after the function name. You cannot pass variables to the function. For example, the following statement executes a function called **error_recovery( )** if an error condition is detected:

```
WHENEVER ERROR CALL error_recovery
```

If you use the BEGIN WORK statement in a function called by WHENEVER, always specify WHENEVER ERROR CONTINUE and WHENEVER WARNING CONTINUE before the ROLLBACK WORK statement. This prevents the program from looping if ROLLBACK WORK encounters an error or warning.

You cannot specify the name of a stored procedure after the CALL keyword. To invoke a stored procedure, use the CALL clause to execute a function that contains an EXECUTE PROCEDURE statement for the desired procedure.

The CONTINUE Option

The CONTINUE keyword to instruct the program to take no action. You can use this keyword to turn off a previously specified option. This is the default option after a WARNING or NOT FOUND condition (and also after an error, if the database is ANSI-compliant).

If you specify WHENEVER ERROR CONTINUE, the built-in START_LOG( ) function does not record errors automatically; see page 4-83.

If the database is ANSI-compliant, CONTINUE is the default action after an error, if no WHENEVER statement is in effect.

The STOP Option

Use the STOP keyword to exit from the program immediately, if the specified exceptional condition occurs. The following statement terminates program execution when the database server issues a warning:

```
WHENEVER WARNING STOP
```

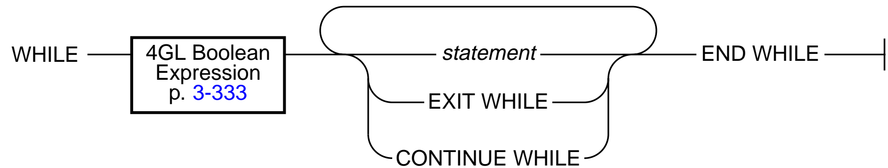
If the database is not ANSI-compliant, the default action after an error is STOP, if no WHENEVER statement is in effect.

References

CALL, DEFER, FOREACH, FUNCTION, GOTO, IF, LABEL, VALIDATE

WHILE

The WHILE statement executes a block of statements while a condition that you specify by a 4GL Boolean expression is TRUE.



statement is an SQL statement or other 4GL statement.

Usage

If the 4GL Boolean expression is TRUE, then 4GL executes the statements that follow the expression, until it encounters the END WHILE keywords. Then it re-evaluates the 4GL Boolean expression.

If the Boolean expression is FALSE, 4GL terminates the loop and resumes execution after the END WHILE keywords. If the Boolean expression is already FALSE on entry to the WHILE statement, then program control passes directly to the statement immediately following the END WHILE keywords.

The following example demonstrates a WHILE loop. If the user responds to the prompt with **y**, 4GL calls the **enter_order( )** function and then prompts the user whether to enter another order. 4GL continues entering orders and prompting the user, as long as the user specifies `y` to the PROMPT:

```
LET answer = "y"
WHILE answer = "y"
    CALL enter_order()
    PROMPT "Do you want to enter another order (y/n) : "
        FOR answer
END WHILE
```

You can interrupt the sequence of statements in a WHILE loop by using the CONTINUE WHILE or EXIT WHILE statement, as described on the next page.

If your database has transaction logging, then it is advisable that the entire WHILE loop be within a transaction (See *Informix Guide to SQL: Tutorial* for more information about the SQL statements that support transactions.)

The CONTINUE WHILE Statement

The CONTINUE WHILE statement interrupts a WHILE loop and cause 4GL to evaluate the Boolean expression again. If the expression is still TRUE, 4GL begins a new iteration of the statements in the loop. If the expression is no longer TRUE, control passes to the statement that follows END WHILE.

The EXIT WHILE Statement

Use the EXIT WHILE statement to terminate the WHILE loop. When the EXIT WHILE keywords are encountered, 4GL does the following:

- Skips all the statements between the EXIT WHILE statement and the END WHILE keywords.
- Resumes execution at the statement following the END WHILE keywords.

The following example demonstrates using the EXIT WHILE statement within a WHILE loop. If the **status** variable is not equal to zero, then 4GL executes the statements that follow the END IF keywords; otherwise, 4GL exits from the WHILE loop and executes the following DISPLAY statement:

```
WHILE TRUE
    ...
    IF status = 0 THEN
        EXIT WHILE
    END IF
    ...
END WHILE
DISPLAY p_customer.* TO customer.*
```

If, as in this example, statements in the WHILE loop cannot change the value of the Boolean expression to FALSE, the WHILE loop cannot terminate unless you specify EXIT WHILE, or GOTO, or some other logical way out of the loop.

The END WHILE Keywords

The END WHILE keywords indicate the end of the WHILE loop, and cause 4GL to evaluate the Boolean expression again. If the expression is still TRUE, 4GL re-executes the statements in the loop. If the expression is no longer TRUE, 4GL passes control to the statement that follows END WHILE.

References

CONTINUE, END, EXIT, FOR, FOREACH

Statement Segments

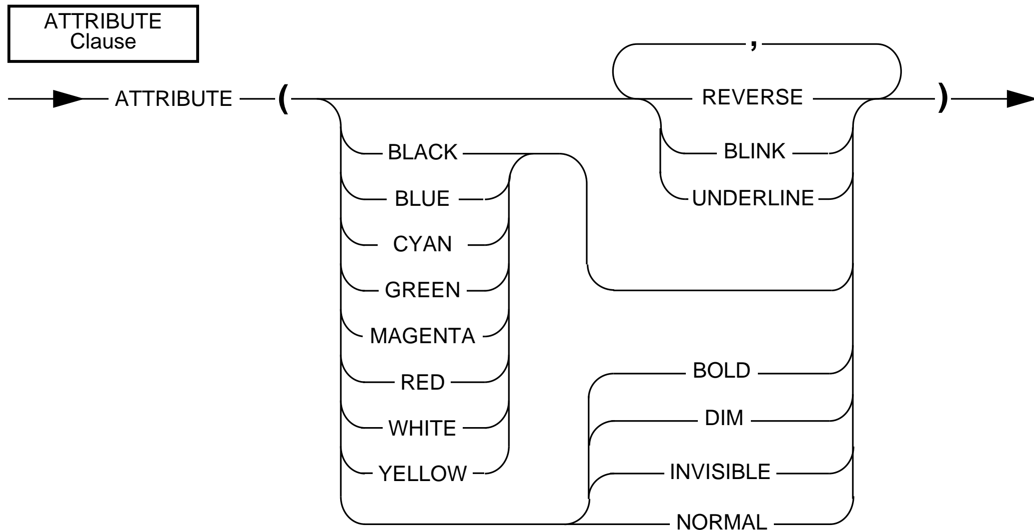
The sections that follow describe *segments* that can appear in the syntax diagrams of some of the 4GL statements that were described in the previous sections.

The following statement segments are described:

- ATTRIBUTE Clause
- Data Types
- Expressions of 4GL
- Field Clause
- Table Qualifiers
- THROUGH, THRU, and .* Notation

ATTRIBUTE

The ATTRIBUTE clause assigns visual attributes in some 4GL statements.



Usage

Keywords listed at the left of this diagram specify *color*; those at the right specify *intensity.* The ATTRIBUTE clause can appear in the following statements:

CONSTRUCT	DISPLAY FORM	INPUT ARRAY
DISPLAY	ERROR	MESSAGE
DISPLAY ARRAY	INPUT	PROMPT

Besides these statements, both the OPEN WINDOW statement (page 3-219) and the OPTIONS statement (page 3-228) can include ATTRIBUTE clauses that support additional keywords, as described earlier in this chapter.

An attribute clause in any statement except OPEN WINDOW or OPTIONS can specify zero or more of the BLINK, REVERSE, and UNDERLINE attributes, and zero or one of the other attributes. That is, all of the attributes except BLINK, REVERSE, and UNDERLINE are mutually exclusive.

Color and Monochrome Attributes

Support for the REVERSE and INVISIBLE attributes does not depend on the color *versus* monochrome status of the monitor. On any monitor, for example, specifying INVISIBLE in an ATTRIBUTE clause prevents its 4GL statement from displaying output on the screen, or else from echoing the user's keystrokes during data entry. (But the screen shows the *character positions* to which the screen cursor moves while the user types.)

For other attributes, 4GL supports either color or monochrome monitors, but not both. If you have a color monitor, you cannot display the *monochrome* attributes (such as BOLD or DIM). If you have a monochrome monitor, you cannot display the *color* attributes (such as RED or BLUE).

For all ATTRIBUTE clauses and field attributes (as described in Chapter 4), the following table shows the effects of the color attributes on a monochrome monitor, as well as the effects of the intensity attributes on a color monitor:

Color Attribute	Monochrome Display	Intensity Attribute	Color Display
WHITE	normal	NORMAL	white
YELLOW	bold	BOLD	red
MAGENTA	bold	DIM	blue
RED	bold		
CYAN	dim		
GREEN	dim		
BLUE	dim		
BLACK	dim		

The following example demonstrates using the ATTRIBUTE clause in an ERROR statement. If the **insert_items( )** function returns FALSE, then 4GL rolls back the changes to the database and displays the error message:

```
IF NOT insert_items( ) THEN
    ROLLBACK WORK
    ERROR "Unable to insert items."
        ATTRIBUTE(RED, REVERSE, BLINK)
    RETURN
END IF
```

If the terminal supports color, 4GL displays the error message in red, blinking, reverse video. If the terminal screen is monochrome, then 4GL displays the error message in bold, blinking, reverse video.

Within its scope (which may be while a field, a form, or a 4GL window is displayed, or while a statement executes), a color attribute overrides any default colors specified for your terminal. (The next page describes the precedence of 4GL attributes.)

Precedence of Attributes

You can assign different attributes to the same field. During execution of field-related statements, however, 4GL uses these rules of precedence (highest to lowest) to resolve any conflicts among attribute specifications:

1. The ATTRIBUTE clause of the current statement.
2. The attributes from the field descriptions in the ATTRIBUTES section of the current form file. (See "Field Attribute Syntax" on page 5-28.)
3. The default attributes specified in the **syscolatt** table of any fields linked to database columns. To modify the **syscolatt** table, use the **upscol** utility. For information on using this utility, see Appendix B.
4. The ATTRIBUTE clause of the most recent OPTIONS statement.
5. The ATTRIBUTE clause of the current form in the most recent DISPLAY FORM statement.
6. The ATTRIBUTE clause of the current 4GL window in the most recent OPEN WINDOW statement.
7. The default reserved line positions and the default foreground color on your terminal.

The field-related statements of INFORMIX-4GL are these:

CONSTRUCT	DISPLAY ARRAY	INPUT
DISPLAY	DISPLAY FORM	INPUT ARRAY

You cannot override the attributes specified for the ERROR, MESSAGE, and PROMPT statements, so precedence rules do not affect these statements.

References

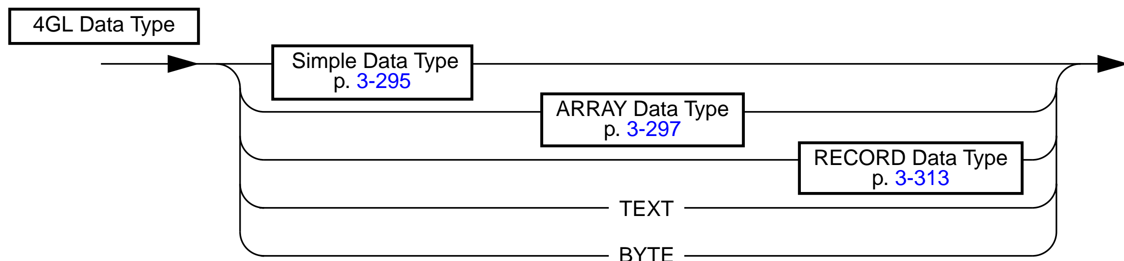
CONSTRUCT, DATABASE, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, ERROR, INPUT, INPUT ARRAY, MESSAGE, OPEN WINDOW, OPTIONS, PROMPT

Data Types of 4GL

You must declare a *data type* for each variable, FORMONLY field, and argument of a function or report. These are the 4GL data types:

Data Type	Kind of Values Stored
ARRAY OF *type*	Arrays of values of any other single data type.
BYTE	Any kind of binary data.
CHAR	Character strings of up to 32,767 ASCII characters.
CHARACTER	(This keyword is a synonym for CHAR.)
DATE	Points in time, specified as calendar dates.
DATETIME	Points in time, specified as calendar dates and time-of-day.
DEC	(This keyword is a synonym for DECIMAL.)
DECIMAL	Fixed point numbers, of a specified scale and precision.
DOUBLE PRECISION	(These keywords are a synonym for FLOAT.)
FLOAT	Floating-point numbers, of up to 32-digit precision.
INT	(This keyword is a synonym for INTEGER.)
INTEGER	Whole numbers, from -2,147,483,647 to +2,147,483,647.
INTERVAL	Spans of time in years and months, or else in smaller time units.
MONEY	Currency amounts, with definable scale and precision.
NUMERIC	(This keyword is a synonym for DECIMAL.)
REAL	(This keyword is a synonym for SMALLFLOAT.)
RECORD	Ordered sets of values, of any combination of **4GL** data types.
SMALLFLOAT	Floating-point numbers, of up to 16-digit precision.
SMALLINT	Whole numbers, from -32,767 to +32,767.
TEXT	Character strings of any length.
VARCHAR	Character strings of varying length, no greater than 255.

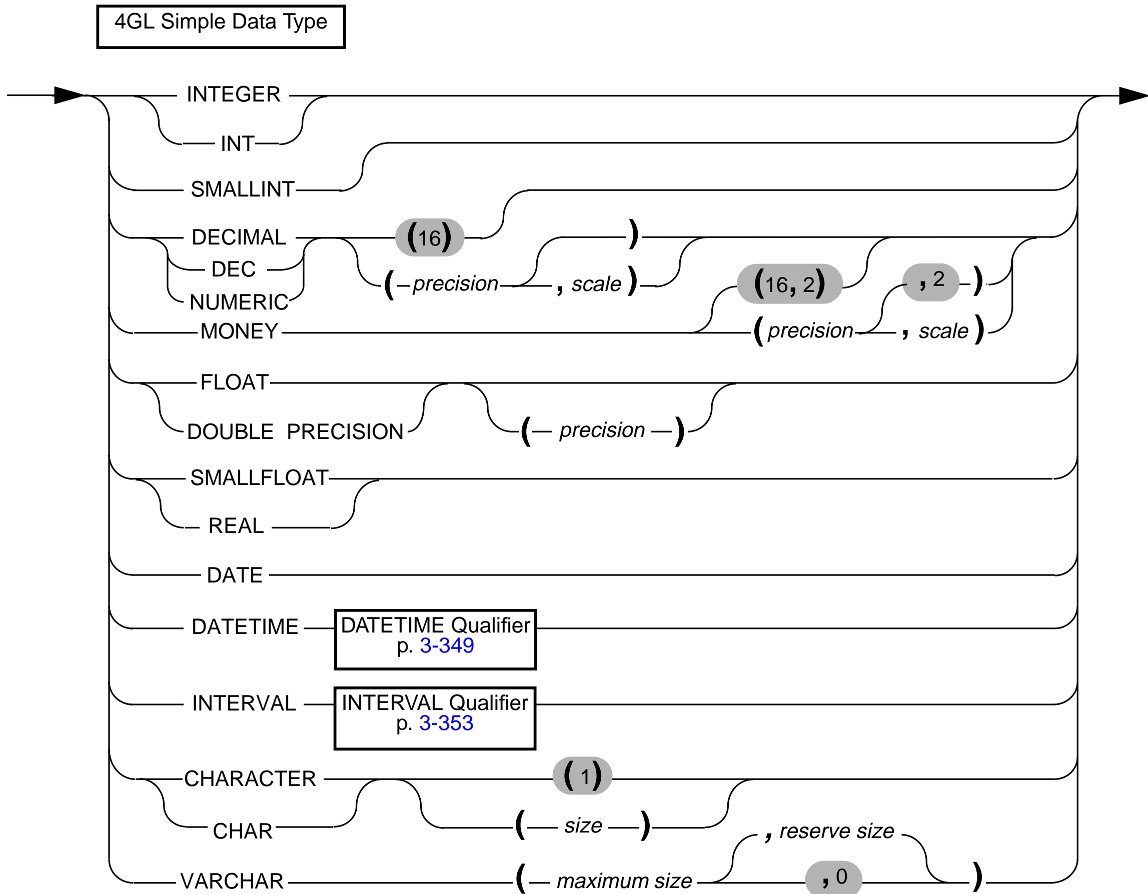
Except for ARRAY and RECORD, each of these corresponds to a data type of SQL. (The SERIAL type of SQL is not a 4GL data type. You can use variables of the INTEGER data type to store SERIAL values from the database.) Every 4GL data type specification uses the following syntax (or else uses the LIKE keyword, as described on page 3-69):



TEXT and BYTE are called the *large* data types; ARRAY and RECORD are called the *structured* data types. All other 4GL data types are *simple* data types.

The Simple Data Types

Each *simple* data type of 4GL can store a single value whose maximum storage requirement is specified or implied in the data type declaration.



maximum size	is a whole number from 1 to 255, specifying the largest number of characters that a VARCHAR variable can store.
precision	specifies the total number of decimal digits, from 1 to 32.
reserve size	is a whole number, from 0 to *maximum size*. The default is 0.
scale	is a whole number, from 1 to *precision*, specifying the number of digits in the fractional part of a fixed-point number.
size	is a whole number from 1 to 32,767, specifying how many characters a CHAR variable stores. The default is 1.

All parameters are literal integers (page 3-340). Sections that follow describe each of these data types. (The articles in this chapter about the FUNCTION, REPORT, and DEFINE statements, as well as Chapter 5 and Chapter 6, describe how to declare the data types of 4GL variables, FORMONLY screen fields, and the formal arguments of functions and reports.)

Each simple data type can be classified as a *number*, *time*, or *character* type:

Number Data Types

4GL supports six simple data types to store various kinds of numbers:

Whole Number Data Types

- SMALLINT Whole numbers, ranging from -32,767 to +32,767.
- INTEGER Whole numbers, from -2,147,483,647 to +2,147,483,647.

Fixed-Point Number Data Types

- DECIMAL (*p,s*) Fixed-point numbers, of scale *s* and precision *p*.
- MONEY Currency values, of a definable scale and precision.

Floating-Point Number Data Types

- DECIMAL (*p*) Floating-point numbers of precision *p* (or by default, 16).
- FLOAT Floating-point, double-precision numbers.
- SMALLFLOAT Floating-point, single-precision numbers.

Time Data Types

4GL supports three simple data types for storing values in chronological units. Two of these store *points in time*, and the third can store *spans of time* (positive or negative differences between points in time):

- DATE Calendar dates (*month*, *day*, *year*) with a fixed scale of *days*, within the range of calendar dates from *January 1 of the year 1* up to *December 31 of the year 9999*.
- DATETIME Instants in time expressed as calendar dates (*year*, *month*, *day*) and time-of-day (*hour*, *minute*, *second*, and *fraction* of a second), within the range of years 1 to 9999.
- INTERVAL Differences between two DATETIME or INTERVAL values, or between a DATE and a DATETIME value, expressed in *years* and *months*, or else in *days* and smaller units of time.

Character Data Types

4GL supports two simple data types for storing *character strings*. Each is designed for a different maximum length of the data string:

- CHAR Strings of up to 32,767 characters. (The ANSI standard also recognizes the keyword "CHARACTER" for this data type.)
- VARCHAR Strings of up to 255 characters, optionally reserving a definable number of bytes (up to 255) of storage space.

Note: *The TEXT data type (described on page 3-317) can contain long text strings, theoretically up to 2^{31} bytes.* INFORMIX-4GL *manipulates TEXT values in a different way, however, from VARCHAR or CHAR values. The INFORMIX-SE engine does not support TEXT or VARCHAR database columns, nor* CHAR *columns of size 32,512 through 32,767 characters. You can, however, declare these data types for* 4GL *variables, formal arguments of functions or reports, and screen fields in forms, regardless of which database engine supports your* 4GL *application.*

The Structured Data Types

4GL supports two *structured* data types for storing sets of values:

- ARRAY Arrays of up to 32,767 values (in any of up to three dimensions) of any data type except ARRAY. (The limit on the total number of elements in an array is compiler-dependent.)
- RECORD Sets of values of any data type, or any combination of types.

The Large Data Types

These store pointers to binary large object ("blob") values, up to 2^{31} bytes in size (or to a limit imposed by your implementation of INFORMIX-OnLine):

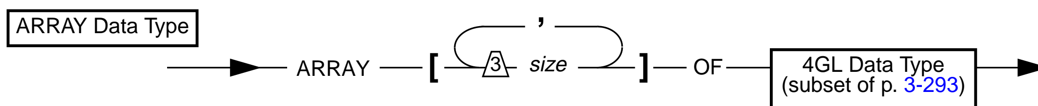
- TEXT Character strings.
- BYTE Anything that can be digitized and stored on your system.

Descriptions of the 4GL Data Types

Sections that follow describe each of the 4GL data types in alphabetical order.

ARRAY

This structured data type stores a 1-, 2-, or 3-dimensional array of variables, all of the same data type. These component variables can be any 4GL data type except ARRAY. This is the syntax for declaring an array of variables:



size is the number (up to 32,767) of elements in a dimension. Each dimension can have a different *size*. (The limit on the total number of elements in an array is imposed by your C compiler, not 4GL.)

Array Elements

Here the data types are a *subset* of those on page 3-293 because ARRAY is not valid. A variable in an array is called an *element*. You can use bracket ([ ]) symbols and comma-separated integers to reference an element of an array. For example, if *array* is the identifier of an array, then:

- *array* [*i*] means the *i*th element of a single-dimensional array.
- *array* [*i, j*] means the *j*th element in the *i*th row of a 2-dimensional array.
- *array* [*i, j, k*] means the *k*th element in the *j*th column of the *i*th row of a 3-dimensional array.

Here *i*, *j*, and *k* can be variables or other integer expressions (page 3-338) that evaluate to positive whole numbers no larger than 32,767.

Note: *Some compilers impose a lower limit on the number of elements in any dimension, or on the total number elements. For an array of two or more dimensions, the total number of elements is the product of all the declared size specifications.*

Substrings of Character Array Elements

If *char_array* [*i, j, k*] is an element of an array of a *character* data type, you can use a comma-separated pair of integer expressions (page 3-338) between trailing bracket ([ ]) symbols to specify a *substring* within its string value:

char_array [*i* , *j* , *k*] [*m* , *n*]

Here $m \leq n$, *for m* and *n* expressions that return positive whole numbers that specify the respective positions of the *first* and *last* characters of a substring within the array element whose coordinates in *char_array* are *i*, *j*, and *k*.

BYTE

The BYTE data type stores any kind of binary data in an undifferentiated byte stream. Binary data typically consist of saved spreadsheets, program load modules, digitized voice patterns, and so on. The INFORMIX-SE database engine does not support BYTE columns, but the 4GL application program can declare program variables of the BYTE data type. The INFORMIX-OnLine engine can store data in BYTE columns.

The data type BYTE has no maximum size; the theoretical limit 2^{31} bytes, but the practical limit is determined by the storage capacity of your system.

You can use a BYTE variable to store, retrieve, or update the contents of a BYTE database column, or to reference a file that you wish to display to users of the 4GL program through an external editor. After you declare a BYTE data type, you must use the LOCATE statement to specify the storage location.

Subsets of BYTE Values

When you select a BYTE column, you can assign all or part of it to a variable of type BYTE. You can use bracket ([ ]) symbols and subscripts to reference only part of a BYTE value, as shown in the following example:

```
SELECT cat_picture [1,75] INTO cat_nip FROM catalog
        WHERE catalog_num = 10001
```

This statement reads the first 75 bytes of the **cat_picture** column of the row with the catalog number **10001**, and stores these data in the **cat_nip** variable.

Restrictions on BYTE Variables

Aggregate functions in reports cannot have BYTE variables as arguments. Among expressions, only 4GL Boolean expressions can reference BYTE variables, and these are valid only when testing for NULL values (page 3-336).

The DISPLAY statement and PRINT statements cannot display BYTE values, nor can the LET statement or INITIALIZE statement assign any value (except NULL) to a BYTE variable. The CALL statement and OUTPUT TO REPORT statements pass any BYTE arguments by reference, not by value.

A form field linked to a BYTE column (or a FORMONLY field of type BYTE) displays the characters `<BYTE value>` rather than actual data. You can use the PROGRAM attribute to display a BYTE value. No other field attributes (except COLOR) can reference the value of a BYTE field. The **upscol** utility cannot set default attributes or values for a BYTE field.

CHAR

The CHAR data type of 4GL can store any ASCII character string, up to a length specified between parentheses (( )) symbols in the *size* parameter of the DEFINE statement that declares the data type. The *size* can range from 1 to 32,767. For example, the variable **keystrokes** in the statement

```
DEFINE keystrokes CHAR(78)
```

can hold a character string of up to 78 characters. If no *size* is specified, as in

```
DEFINE keystrokes CHAR
```

the resulting *default* CHAR data type can store only a single character. In a form, you cannot specify the *size* of a FORMONLY CHAR field; the *size* defaults to the field length from the screen layout. On the INFORMIX-SE engine, the maximum data string length that a CHAR column can store is 32,511.

The CHAR data type requires one byte of storage per character, or *size* bytes. When a string value is passed between a CHAR variable and a CHAR database column, or between two CHAR variables, exactly *size* bytes of data are transferred, for *size* the declared length of the 4GL variable or the database column that receives the string. If the length of the data string is shorter than this *size*, the string is padded with trailing blank spaces to fill the declared *size*. If the string is longer than *size,* then the stored value is truncated. A character string returned by a function can contain no more than 511 bytes of data.

To perform arithmetic on numbers stored in variables, use a *number* data type. CHAR variables can store digits, but you might not be able to use them in some calculations. Similarly, leading zeros (as in some postal codes) are stripped from values stored as the number data types INTEGER or SMALLINT. You should store such values as CHAR data types.

NLS

When accessing an NLS database in the standard way (Implicit NLS), all character data in the database is sorted and compared according to the locale established at the time of database creation. For example, the 4GL user cannot create character data columns that sort in US English if the database locale (the locale the database was created in) is non-US-English. Although character columns defined by a 4GL program are defined within the program as type CHAR (or VARCHAR, if variable-length), they are interpreted by the server as type NCHAR (or NVARCHAR) if the database is non-US-English. This process is known as *implicit mapping.*

4GL supports the Implicit and Open NLS environments. Explicit NLS is not supported in **4GL**. For more information, see Appendix E, "Native Language Support Within INFORMIX-4GL."

CHARACTER

The CHARACTER keyword is a synonym for CHAR.

DATE

The DATE data type stores calendar dates. A calendar date is stored internally as an integer that evaluates to the number of days since December 31, 1899. The default display format of a DATE value is

mm/dd/yyyy

for *mm* a *month* (1 to 12), *dd* a *day* of the month (1 to 31), and *yyyy* a *year* (0001 to 9999). The DBDATE environment variable can change the separator symbol

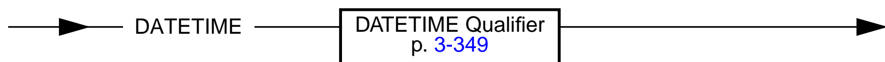
and the default order of *day*, *month*, and *year* time units. The FORMAT attribute specifies DATE display formats in forms. For *month*, 4GL accepts the value `1` or `01` for January, `2` or `02` for February, and so on. For *days*, it accepts a value `1` or `01` for the first day of the month, `2` or `02` for the second, and so on, up to the maximum number of days in a given month.

If the user of your application enters from the keyboard a two-digit value for the *year*, as in `89` or `93`, then 4GL assumes that the year is in the range 1900-1999, and assigns `1` and `9` as the first two digits (`19`) of the year. (Users must pad the *year* value with one or two leading zeros to specify years in the First Century; for example, "`093`" or "`0093`" for the year 93 A.D.)

Because DATE values are stored as integers in the range 693,594 to 2,958,464, you can use them in arithmetic expressions, such as the difference between two DATE values. The result, a positive or negative INT value, is the number of days that have elapsed between the two dates. The UNITS DAY operator can convert this to an INTERVAL. (The use of DATE operands in division, multiplication, or exponentiation cannot produce meaningful results.)

DATETIME

The DATETIME data type stores an instant in time, expressed as a calendar date and time-of-day. You specify the time units that the DATETIME value stores; the precision can range from a year through a fraction of a second. Data are stored as DECIMAL numbers that represent a contiguous sequence of values, each representing units of time. Its declaration uses this syntax:



The scale and precision specification is called the *DATETIME qualifier*. It uses a *"first* TO *last"* format to declare variables and screen fields. You must substitute one or two of these keywords for the *first* and *last* terms:

Keyword	Corresponding Time Unit and Range of Values
YEAR	A *year*, numbered from 0001 (A.D.) to 9999.
MONTH	A *month*, numbered from 1 to 12.
DAY	A *day*, numbered from 1 to 31, as appropriate for its month.
HOUR	An *hour*, numbered from 0 (midnight) to 23.
MINUTE	A *minute*, numbered from 0 to 59.
SECOND	A *second*, numbered from 0 to 59.
FRACTION (*scale*)	A decimal *fraction of a second*, with a *scale* of up to five digits.
or FRACTION	The default *scale* is three digits (thousandth of a second).

The keyword specifying the *last* term cannot represent a larger unit of time than the *first* keyword. Thus, YEAR TO SECOND or HOUR TO HOUR are valid, but DAY TO MONTH results in a compiler error, because the *last* keyword (here MONTH) specifies a larger unit of time than DAY, the *first* keyword.

Unlike INTERVAL qualifiers, DATETIME qualifiers *cannot* specify non-default precision (except for FRACTION, if that is the *last* keyword in the qualifier). The following are examples of valid DATETIME qualifiers:

```
DAY TO MINUTE YEAR TO MINUTE
FRACTION TO FRACTION(4)MONTH TO SECOND
```

Operations with DATETIME values that do not include YEAR in their qualifier use the system date to supply any additional precision. If the *first* term is DAY and the current month has fewer than 31 days, unexpected results can occur.

For example, assume that it is February, and you wish to store data from January 31 in the **sometime** variable that is declared in the next statement:

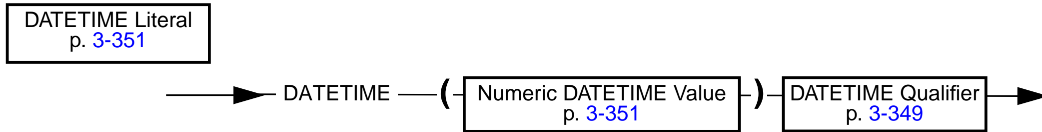
```
DEFINE sometime DATETIME DAY TO MINUTE
CREATE TABLE mytable (mytime DATETIME DAY TO MINUTE)
LET sometime = DATETIME(31 12:30) DAY TO MINUTE
INSERT INTO mytable VALUES (sometime)
```

Because the column **mytime** does not store the *month* nor *year*, the current month and year are used to evaluate whether the inserted value is within acceptable bounds. February has only 28 (or 29) days, so no value for DAY can be larger than 28 (or 29 some years). The INSERT statement in this case

would fail, because the value `31` for *day* is out of range for February. To avoid this problem, qualify DATETIME data types with YEAR or MONTH as the *first* keyword, and do not enter data values with *day* as the largest time unit.

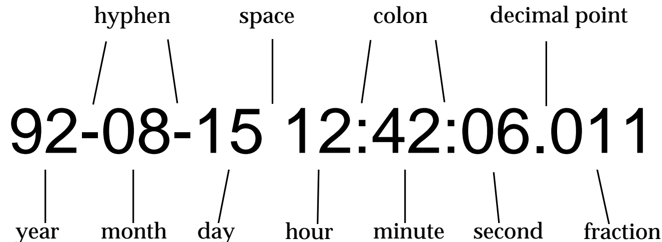
DATETIME Literals and Delimiters

Statements of 4GL can assign values to DATETIME data types. The simplest way to do this is as a DATETIME literal or as a character string. Both formats represent a specific DATETIME value as a *numeric DATETIME value*.



The *DATETIME literal* format begins with the DATETIME keyword, followed by a pair of parentheses that enclose unsigned whole numbers (separated by delimiters) to represent a consecutive sequence of *year* through *fraction* values, or a subset thereof. This must be followed by a DATETIME qualifier, specifying the *first* TO *last* keywords for the set of time units in the numeric DATETIME value.

For example, DATETIME YEAR TO FRACTION(3) values require six delimiters:



These are the delimiters that are required within DATETIME literal values.

Delimiter	Position within DATETIME Value
hyphen (**-**) symbol	Between the *year*, *month*, and *day* portions of the value.
blank ( ) space	Between the *day* and *hour* portions
colon (**:**) symbol	Between the *hour*, *minute*, and *second* portions.
decimal (**.**) point	Between the *second* and *fraction* portions.

DATETIME literals can specify every time unit from the data-type declaration, or only the units of time that you need. For example, you can assign a value qualified as MONTH TO HOUR to a variable declared as YEAR TO MINUTE, if the value contains information for a contiguous sequence of time units. You cannot, however, assign a value for just *month* and *hour*; in this case, the DATETIME literal must also include a value (and delimiters) for *day*.

A DATETIME literal that specifies fewer units of time than in the declaration is automatically expanded to fill all the declared units. If the omitted value is for the *first* unit of time, or for this and for other time units *larger* than the largest unit that is supplied, then the missing units are automatically supplied from the system clock-calendar. If the value omits any *smaller* time units, their values each default to zero (or to `1` for *month* and *day*) in your entry. To specify a year between 1 and 99, you must pad the *year* value with leading zeros.

Character Strings as DATETIME Values

You can also specify a DATETIME value as a *character string*, indicating the numeric values of the date and time. In a 4GL source module, this must be enclosed between a pair of quotation (") marks, without the DATETIME keyword and without qualifiers, but with all the required delimiters. Unlike DATETIME literals, the character string must include information for *all* the units of time declared in the DATETIME qualifier. For example, the following LET statement specifies a DATETIME value entered as a character string:

```
LET call_dtime = "1992-08-14 08:45"
```

In this case, the **call_dtime** variable was declared as DATETIME YEAR TO MINUTE, so the character string must specify values for *year*, *month*, *day*, *hour*, and *minute* time units. If the character string does not contain information for all the declared time units, an error results. Similarly, an error results if a delimiter is omitted, or if extraneous blanks appear within the string.

When a user of the 4GL program enters data in a DATETIME field of a screen form, or during a PROMPT statement that expects a DATETIME value, the only valid format is a numeric DATETIME, entered as an unquoted string. Any entry in a DATETIME field must be a contiguous sequence of values for *units of time* and *delimiters*, in the following format (or in some subset of it):

year-month-day hour:minute:second.fraction

Depending on the data type declaration of the DATETIME field, each of these *units of time* can have values that combine traditional base-10, base-24, base-60, and lunar calendar values from clocks and calendars.

Values that users enter in a DATETIME field of the 4GL form need not include all the declared time units, but users *cannot* enter data as DATETIME literals, a format that is valid only within 4GL statements and in the data type declarations of FORMONLY DATETIME fields of form specification files.

All time-unit values of a DATETIME data type are two-digit numbers, except for the *year* and *fraction* values. The *year* is stored as four digits. The *fraction* requires *n* digits, for $1 \leq n \leq 5$, rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required to store a DATETIME value:

((total number of digits for all time units)/ 2) + 1

For example, a YEAR TO DAY qualifier requires a total of eight digits (four for *year*, two for *month*, and two for *day*), or ((8/2) + 1) = 5 bytes of storage. For information on using DATETIME values in time expressions and in 4GL Boolean expressions, see the section "Expressions of 4GL" on page 3-326.

DEC

The DEC keyword is a synonym for DECIMAL.

DECIMAL

The DECIMAL data type stores values as decimal numbers, up to 32 significant digits. As the declaration syntax for simple data types on page 3-293 indicates, you can optionally specify the *precision* (the number of significant digits) and the *scale* (the number of digits to the right of the decimal point). For example, "DECIMAL (14,2)" specifies a total of 14 significant digits, two of which describe the fractional part of the value.

When you specify both the precision and scale, the 4GL program can manipulate the DECIMAL variable or constant with fixed-point arithmetic.
If the data type declaration specifies neither the precision nor the scale, the default is DECIMAL(16), a floating-point decimal with a precision of 16 digits. If you specify only one parameter, this is interpreted as the precision of a floating-point number that can range in absolute value from 10^{-128} to 10^{126}.

The largest absolute value that you can store without an error in a DECIMAL variable is $10^{p-s} - 10^{-s}$. Here *p* is the precision, and *s* is the scale. Values with an absolute value less than 0.5×10^{-s} are stored as zero. You cannot specify *p* nor *s* for a FORMONLY DECIMAL field in a form; its precision is the *smaller* of 32 and (*length* - 2), where *length* is the field width in the screen layout.

DECIMAL (*p,s*) data types are useful for storing numbers with fractional parts that must be calculated exactly, such as rates or percentages. Unless you are developing a scientific or engineering application that explicitly controls for measurement error, store floating-point numbers as DECIMAL(*p,s*) values.

When a user enters data in a SMALLFLOAT or FLOAT field, 4GL converts the base-10 value to binary format for storage. Likewise, to display a FLOAT or SMALLFLOAT number, 4GL reformats it from binary to base-10. Both conversions can lead to inaccuracy. Thus, if a user enters `10.7` into a FLOAT field, it might actually be stored as `10.699999` or as `10.700001`, depending on the magnitude and the sign of the binary-to-decimal conversion error. This limitation is a feature of digital computers, rather than of 4GL.

The following formulae calculate the storage needed (in bytes) for DECIMAL values; you must round any fractional result up to the next whole number:

When scale is EVEN: (*precision* + 3) / 2)
When scale is ODD: (*precision* + 4) / 2)

For example, DECIMAL(15,2) requires ((15 + 3) / 2) = 9 bytes of storage.

DOUBLE PRECISION

The DOUBLE PRECISION keywords are a synonym for FLOAT.

FLOAT

The FLOAT data type stores values as double-precision floating-point binary numbers with up to 16 significant digits. FLOAT corresponds to the *double* data type in the C language. Values for the FLOAT data type have the same range of values as the C *double* data type on your C compiler.

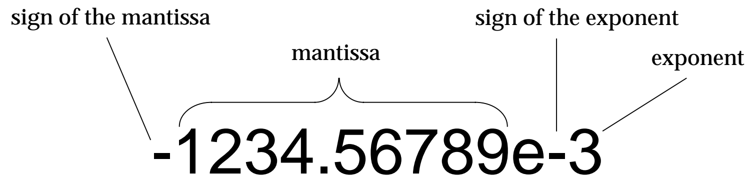
For compatibility with the ANSI standard for embedded SQL, you can declare a whole number between 1 and 14 as the precision of a FLOAT data type, but the actual precision is data-dependent and compiler-dependent.

A variable or constant of the FLOAT data type typically stores scientific or engineering data that can only be calculated approximately. Since floating-point numbers retain only their most significant digits, a value that is entered into a FLOAT variable, constant, or database column can differ slightly from the numeric value that a 4GL form or report displays.

This rounding error arises from how computers store floating-point numbers internally. For example, you might enter a value of `1.1` into a FLOAT field. After processing the 4GL statement, the program might display this value as `1.09999999`. This occurs in the typical case where the exact floating-point

binary representation of a base-10 value requires an infinite number of digits in the mantissa. The computer stores a finite number of digits, so it stores an approximate value, with the least significant digits treated as zeros.

Statements of 4GL can specify FLOAT values as *floating-point literals*:



You can use uppercase or lowercase `E` as the exponent symbol; omitted signs default to + (positive). If a number in another format (such as an integer or a fixed-point decimal) is supplied in a **.4gl** file or from the keyboard when a FLOAT value is expected, 4GL attempts data type conversion.

FLOAT data types usually require 8 bytes of memory storage. In reports and screen displays, you can use the USING operator to format FLOAT values. Unless you do so, the default scale in output is two (2) decimal digits.

INT

The INT keyword is a synonym for INTEGER.

INTEGER

The INTEGER data type stores whole numbers in a range from -2,147,483,647 to +2,147,483,647. The negative number `-2,147,483,648` is a reserved value that cannot be used. INTEGER values are stored as signed four-byte binary integers, with a scale of zero. This data type is typically used to stores counts, quantities, categories coded as natural numbers, and the like.

Arithmetic operations on binary integers are typically without rounding error; these operations and sort comparisons are performed more efficiently than on FLOAT or DECIMAL data. INTEGER values, however, can only store data whose absolute value is less than 2^{31}. Any fractional part of the value is discarded. If a value exceeds this numeric range, neither 4GL nor the database engine can store the data value as an INTEGER data type.

Note: *INTEGER variables can store SERIAL values from the database. If a user inserts a new row into the database, 4GL automatically assigns the next whole number in sequence to any field linked to a SERIAL column. Users do not need to enter data into such fields. Once assigned, a SERIAL value cannot be changed.*

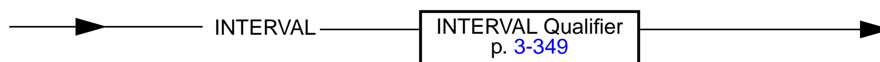
INTERVAL

This data type stores *spans of time*, the differences between two points in time. You can also use it to store quantities that are naturally measured in units of time, such as age or sums of ages, estimated or actual time required for some activity, or person-hours or person-years of effort attributed to some task.

An INTERVAL data value is stored as a DECIMAL number that includes a contiguous sequence of values representing units of time. The INTERVAL data types of 4GL fall into two classes, based on their precision:

- A *year-month interval* can represent a span of *years* and/or *months*
- A *day-time interval* can represent a span of *days*, *hours*, *minutes*, *seconds*, and *fractions* of a second, or a contiguous subset of those time units.

Unlike DATETIME data types, which they somewhat resemble in their format, INTERVAL data types can assume zero or negative values. The declaration of an INTERVAL data type uses the following syntax:



INTERVAL Qualifiers

The *INTERVAL qualifier* specifies the precision and scale of an INTERVAL data type, using a *"first* TO *last"* format to declare 4GL variables, named constants, function and report parameters, and screen fields. It has the same syntax in declarations of 4GL variables, named constants, and FORMONLY fields as for INTERVAL columns of the database.

You must substitute one or two keywords from only one the following lists for the *first* and *last* keywords of an INTERVAL qualifier:

YEAR-MONTH INTERVAL Keywords	DAY-TIME INTERVAL Keywords
YEAR, MONTH	DAY, HOUR, MINUTE, SECOND, FRACTION

As with DATETIME data types, you can declare INTERVAL data types to include only the units that you need. INTERVAL represents a span of time independent of an actual date, however, so you *cannot* mix keywords from both lists in the same INTERVAL qualifier. Since the number of days in a month depends on which month it is, an INTERVAL data value cannot combine both *months* and *days* as units of time. For example, specifying "MONTH TO MINUTE" as an INTERVAL qualifier would results in a compile-time error.

For any *first* keyword except FRACTION, you have the option of specifying a *precision* of up to nine digits; otherwise the default precision is 2 digits, except for YEAR, which defaults to four digits of precision. If an INTERVAL qualifier specifies only a single unit of time, the *first* and *last* keywords are the same. When the *first* and *last* keywords are both FRACTION, you can only specify the *scale* after the *last* keyword.

When the *last* keyword is FRACTION, you can specify a *scale* of 1 to 5 digits; otherwise, the *scale* defaults to three digits (thousandth of a second). For example, the following are valid INTERVAL qualifiers:.

```
HOUR TO MINUTE          MONTH(5) TO MONTH
FRACTION TO FRACTION(4) HOUR(9) TO FRACTION(5)
```

The keyword specifying the *last term* cannot represent a larger unit of time than the *first* keyword. Thus, YEAR TO MONTH or HOUR TO HOUR are valid, but HOUR TO DAY results in a compiler error, because the *first* keyword (here HOUR) specifies a smaller unit of time than DAY, the *last* keyword.

After you declare an INTERVAL data type, a 4GL statement can assign it the value of a *time expression* (page 3-347) that specifies an INTERVAL value. The simplest way to do this is as an INTERVAL literal or as a character string. Both formats require that you specify a numeric INTERVAL value.

INTERVAL Literals and Delimiters

The *INTERVAL literal* format begins with the INTERVAL keyword, followed by a pair of parentheses that enclose unsigned whole numbers (separated by delimiters) to represent a consecutive sequence of *year* through *fraction* values, or as a portion thereof. This must be followed by a valid INTERVAL qualifier, specifying the *first* TO *last* keywords for the set of time units:

INTERVAL Literal
p. 3-355

Numeric INTERVAL values use the same delimiters as DATETIME values, except that *month* and *day* need no separator, since they cannot both appear in the same INTERVAL value. Any time unit value in a numeric INTERVAL, has a default precision of two digits, except for *year*, which defaults to four digits, and *fraction*, which defaults to three. The INTERVAL qualifier can override these defaults for the *first* time unit, and for the scale of the *fraction*.

The following delimiters are required in INTERVAL values.

Delimiter	Position within INTERVAL Value
hyphen (**-**) symbol	Between the *year*, *month*, and *day* portions of the value.
blank ( ) space	Between the *day* and *hour* portions.
colon (**:**) symbol	Between the *hour, minute*, and *second* portions.
decimal (**.**) point	Between the *second* and *fraction* portions.

For example, INTERVAL YEAR(3) TO MONTH values require one delimiter:

hyphen

year 102-08 month

Similarly, INTERVAL DAY(6) TO FRACTION(2) values require four delimiters

space colon decimal point

120815 12:42:06.01

day hour minute second fraction

INTERVAL literals can specify all the time units from the data type declaration, or only the units that you need. For example, you can assign a value qualified as HOUR TO MINUTE to a variable declared as DAY TO SECOND, if the entered value contains information for a contiguous sequence of time units. You cannot, however, assign only *day* and *minute* values; in this case, the INTERVAL literal must also include a value (and delimiters) for *hour*.

The value for the *first* time units in an INTERVAL literal can have up to nine digits of precision (except for FRACTION, which cannot include more than five digits). If a *first* unit value to be entered is greater than the default number of digits for that time unit, however, you must explicitly identify the number of significant digits that you are entering. For example, to specify an INTERVAL of DAY TO HOUR that spans 162.5 days, you can use the format

```
INTERVAL (162 12) DAY(3) TO HOUR
```

To specify an INTERVAL literal in a 4GL statement, you must include numeric values for both the *first* and *last* time units from the qualifier of the INTERVAL literal, and also values for any intervening time units.

You can optionally specify the *precision* of the *first* time unit (and also a *scale*, if the *last* keyword of the INTERVAL qualifier is FRACTION).

Character Strings as INTERVAL Values

You can also specify an INTERVAL value as a *character string*, indicating the numeric values of the time units. In a 4GL source code module, this must be enclosed between a pair of quotation(") marks, without the INTERVAL keyword and without qualifiers, but with all the required delimiters. Unlike INTERVAL literals, the character string must include information for *all* the units of time declared in the INTERVAL qualifier. For example, the character string in the next statement specifies a span of five years and six months:

```
LET long_time = "5-06"
```

Similarly, values entered as character strings into INTERVAL columns of the database must include information for *all* time units that were declared for that column. For example, the following INSERT statement shows an INTERVAL value entered as a character string:

```
INSERT into manufact (manu_code, manu_name, lead_time)
    VALUES ("BRO", "Ball-Racquet Originals", "160")
```

Since the **lead_time** column is defined as INTERVAL DAY(3) TO DAY, this INTERVAL value requires only one value, indicating the number of days required. If the character string does not contain information for all the declared time units, the database engine returns an error.

Data Entry by Users

When a user of the 4GL program enters data in an INTERVAL field of a form, the only valid format is as an *unquoted* character string. Any entry into an INTERVAL field must be a contiguous sequence of values for *units of time* and *separators*, in one of these two formats (or in some subset of one):

year **-** *month*
day *hour* **:** *minute* **:** *second* **.** *fraction*

Depending on the data type declaration of the INTERVAL field, each of these *units of time* (except the *first*) is restricted to values that combine traditional base-10, base-24, base-60, and lunar calendar values from clocks and calendars. The *first* value can have up to nine digits, unless FRACTION is the first unit of time. (If FRACTION is the *first* time unit, the maximum *scale* is 5 digits.)

Values that users enter in a INTERVAL field of a 4GL form need not include all the declared time units, but users *cannot* enter data as INTERVAL literals, a format that is valid only within 4GL statements and in data type declarations of FORMONLY fields of data type INTERVAL in form specification files.

By default, all values for time units in a numeric INTERVAL are two-digit numbers except for the *year* and *fraction* values. The *year* value is stored as four digits. The fraction value requires *n* digits where $1 \leq n \leq 5$, rounded up to an even whole number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required for an INTERVAL value:

total number of digits for all time units)/2) + 1

For example, a YEAR TO MONTH qualifier requires a total of six digits (four for *year* and two for *month*), or ((6/2) + 1) = 4 bytes of storage.

For information on using INTERVAL data in arithmetic and relational operations, see the section "Expressions of 4GL" on page 3-326.

MONEY

The MONEY data type stores currency amounts. Like the DECIMAL data type, the MONEY data type stores fixed-point numbers, up to a maximum of 32 significant digits. As the syntax diagram for simple data type declarations indicates (page 3-293), you can optionally include one or two whole numbers to specify the *precision* (the number of significant digits) and the *scale* (the number of digits to the right of the decimal point).

Unlike the DECIMAL data type (page 3-304), which stores floating-point numbers if its data-type declaration specifies neither scale nor precision, MONEY values are always stored as fixed-point decimal numbers. If you declare a MONEY data type with only one parameter, then 4GL interprets that parameter as the precision. By default, the scale is 2, so the data type MONEY(*p*) is stored internally as DECIMAL(*p,2*), for *p* the precision ($1 \leq p \leq 32$). If no parameters are specified, MONEY is interpreted as DECIMAL(16,2). This specifies a total of 16 significant digits, two of which describe the fractional part of the currency value.

The largest absolute value that you can store without error as a MONEY data type is $10^{p-s} - 10^{-s}$. Here *p* is the precision, and *s* is the scale. Values with an absolute value less than 0.5×10^{-s} are stored as zero. You cannot specify the precision nor the scale of a FORMONLY MONEY field in an 4GL form; here the precision defaults to the smaller of 32 or (*length* - 2), for *length* the field length from the SCREEN section layout.

On the screen, MONEY values are displayed with a currency symbol, by default, a dollar ($) sign, and a decimal (.) point. You can change the default display format for MONEY values by changing the DBMONEY environment variable. 4GL statements and keyboard input by users to fields of screen forms do *not* need to include currency symbols in literal MONEY values.

The same formulae as for DECIMAL values (page 3-304) apply to MONEY data types; round any fractional result up to the next whole number:

When scale is EVEN: (*precision* + 3) / 2 ; *When scale is ODD:* (*precision* + 4) / 2

For example, a MONEY(13,2) variable has a precision of 13 and a scale of 2. This requires ((13 + 3) / 2) = 8 bytes of storage.

NUMERIC

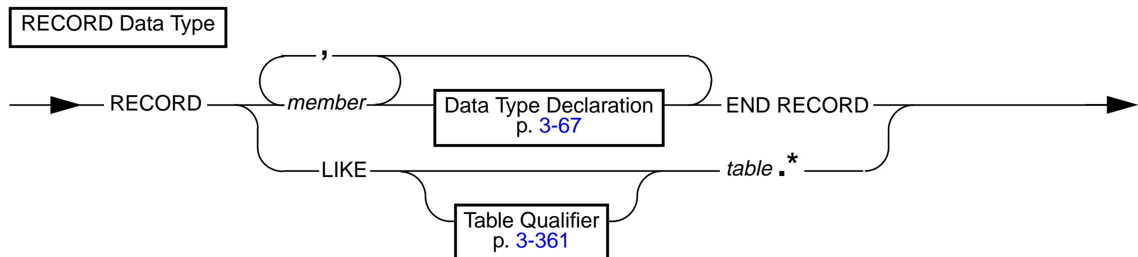
The NUMERIC keyword is a synonym for DECIMAL. (When the word *numeric* appears in lowercase characters in this manual, it is always the adjective formed from the noun "number," rather than the name of a data type.)

REAL

The REAL keyword is a synonym for SMALLFLOAT. (When the phrase *real number* appears in lowercase characters in this manual, it denotes a number that is neither imaginary nor transfinite, rather than the name of a data type.)

RECORD

The RECORD data type stores an ordered set of values. Within each of these sets (called a *program record*), the values can be of any 4GL data type, or any combination of data types in a fixed order, including the *simple* data types (page 3-295), *large binary* data types (BYTE, TEXT), and *structured* data types (ARRAY, RECORD). A component variable in a record is called a *member*. This is the data type declaration syntax for RECORD variables:



member is a name that you declare for a member variable of the record.

table is the SQL identifier of a database table, synonym, or view.

You can use the LIKE keyword after the name of a member variable to specify that the variable the same data type as some column of a database table. If you do not specify explicit member names, and use an asterisk (*) symbol

after a *table* name, you declare a record with members that have the same identifiers as the columns in *table*; their 4GL data types correspond to the fixed sequence of SQL data types in an entire row of the *table*. (Any SERIAL column in *table* corresponds to a record member of data type INTEGER.)

This example uses the LIKE keyword to declare two program records, one of which contains a member variable of the RECORD data type called **nested**:

```
DEFINE
    p_customer RECORD LIKE informix.customer.*,
    p_orders RECORD
            order_num LIKE informix.orders.order_num,
            nested RECORD a LIKE informix.items.item_num,
                         b LIKE informix.stock.unit_descr
                   END RECORD
            END RECORD
```

If *table* is a view, the *column* cannot be based upon an aggregate. You cannot specify *table*.* if *table* is a view that contains an aggregate column.

In an ANSI-compliant database, you must qualify the *table* name with the *owner* prefix, if the program will be run by users other than *owner*. If the *table* is an *external* or *external, distributed* table, its name must be qualified by the name of the remote database and by the name of the database server.

Referencing Record Members

If record is the identifier that you declare for a program record in a DEFINE statement, or the name of a *screen record* in a 4GL form, you can use the following notation to reference members of the record in 4GL statements:

- The notation *record.member* refers to an individual member of a record, where *member* is the identifier of the member.
- The notation *record.first* THRU *record.last* refers to a consecutive subset of members, from *record.first* through *record.last* inclusive. Here *first* is an identifier that was listed before *last* among the explicit or implicit member names in the RECORD declaration; see page 3-363. You can also use the keyword THROUGH as a synonym for THRU.
- The notation *record.** refers to the entire record.

Several restrictions apply when you reference members of a record:

- You cannot use THRU nor THROUGH to indicate a partial list of *screen record* members when displaying or entering data in a 4GL screen form.
- You cannot use the THRU, THROUGH, nor .* to reference a program record that contains an *array* among its members. (But you can use these to specify all or part of a record that contains one or more records as members.)
- You cannot use THRU, THROUGH, nor .* notation in a SELECT or INSERT *variable list* in a *quoted string* within the PREPARE statement. (You can, however, use the .* notation to specify a program record in the *variable list* of an INSERT or SELECT clause of the DECLARE statement.)

A program record whose members correspond in number, order, and data type compatibility to a database table or to a screen record (as described in Chapter 6) can be useful for transferring data from the database to the screen, to reports, or to functions of the 4GL program.

SMALLFLOAT

The SMALLFLOAT data type stores values as single-precision floating-point binary numbers, with up to 8 significant digits. The range of SMALLFLOAT values is the same as for the *float* data type on your C compiler. The storage requirement is usually 4 bytes. The SMALLFLOAT data type typically stores scientific or engineering data that can only be calculated approximately. Since floating-point numbers retain only their most significant digits, a value that is entered into a SMALLFLOAT variable, constant, or column can differ slightly from the base-10 value that an 4GL form or 4GL report displays.

This error arises from the internal storage format of binary floating-point numbers. For example, if you enter a value of 1.1 into a SMALLFLOAT field and, after processing the 4GL statement, the screen might display this value as `1.1000001`. This occurs in the typical case where the exact floating-point binary representation of a base-10 value requires an infinite number of digits in the mantissa. A computer stores only a finite number of digits, so it stores an approximate value, with the least-significant digits treated as zeros.

Statements of 4GL can specify SMALLFLOAT values as *floating-point literals*, using the following format:

You can use uppercase or lowercase `E` as the exponent symbol; omitted signs default to + (positive). If a literal value in another format (such as an integer or a fixed-point decimal) is supplied from the keyboard into a SMALLFLOAT field, or in a statement, 4GL attempts data type conversion.
fixed-point decimal) is supplied in a .4-point decimal) is supplied from the keyboard into a SMALLFLOAT field, or in a statement, 4GL attempts data type conversion.

In reports and screen displays, the USING operator can format SMALLFLOAT values. Unless you do so, the default scale in output is two (2) digits.

SMALLINT

The SMALLINT data type stores data as signed two-byte binary integers. Values must be whole numbers within the range from -32,767 to +32,767. Any fractional part of the data value is discarded. If a value lies outside this range, you cannot store it in a SMALLINT variable or database column. (The negative value -32,768 is reserved; it cannot be assigned to a SMALLINT variable, constant, or database column, nor entered into a SMALLINT field.)

You can use SMALLINT variables, constants, and FORMONLY fields of screen forms to store, manipulate, and display data that can be represented as whole numbers of an absolute value less than 2^{15}. This data type typically stores small whole numbers, Boolean values, ranks, 4-digit codes, or measurements that classify data into a small number of numerically-coded categories. Since a SMALLINT data type requires only 2 bytes of storage, arithmetic operations in number expressions can be done very efficiently, provided that the values of the operands lie within the somewhat limited range of SMALLINT values.

TEXT

The TEXT data type stores character data in ASCII strings. TEXT resembles the BYTE data type, but 4GL supports features to display TEXT variables whose values are restricted to combinations of *printable* ASCII characters and the following control characters:

- TAB (= CONTROL-I)
- NEWLINE (= CONTROL-J)
- FORMFEED (= CONTROL-L)

Note: *If you include other non-printable characters in a TEXT string, the features of 4GL for displaying TEXT values may not operate correctly; see page 3-345.*

Character strings stored as TEXT variables have a theoretical limit of 2^{31} bytes, and a practical limit determined by the available storage on your system. The INFORMIX-SE database engine does not support TEXT columns, but regardless of the engine, you can declare 4GL variables of type TEXT.

You can use a TEXT variable to store, retrieve, or update the contents of a TEXT database column, or to reference a file that you wish to display to users of the 4GL program through a text editor. After you declare a TEXT data type, you must use the LOCATE statement to specify the storage location.

When you retrieve a value from a TEXT column, you can assign all or part of it to a variable. Use bracket ([ ]) symbols and comma-separated subscripts to reference only a specified part of a TEXT value, as in the following example:

```
SELECT cat_description [1,75] INTO cat_nap FROM catalog
        WHERE catalog_num = 10001
```

This reads the first 75 bytes of the **cat_description** column of the row with the catalog number **10001**, and stores these data in the **cat_nap** program variable.

Restrictions on TEXT Variables

In an 4GL form, a field linked to a TEXT column (or a FORMONLY field of type TEXT) only displays as many characters as can fit in the field. To display TEXT values longer than the screen field, or to edit a TEXT value, you must assign the PROGRAM attribute to the TEXT field. The WORDWRAP attribute can display the initial characters of a TEXT value, up to the last segment of the field, but cannot edit a value in a TEXT field. No other 4GL field attribute (except COLOR) can reference the value of a TEXT field.

In a CALL statement, TEXT arguments are passed by reference, rather than by value (page 3-18). You cannot use the DISPLAY statement to display a TEXT value. You cannot use the LET statement to assign any value (except NULL) to a TEXT variable.

Aggregate report functions (page 6-46) cannot specify TEXT values as arguments. In 4GL expression (page 3-326), you can only reference TEXT variables to test for NULL values, or as an operand of the WORDWRAP operator.

VARCHAR

The VARCHAR data type stores character strings of varying length. As the syntax diagram for simple data type declarations indicates (page 3-293), you can optionally specify the *maximum size* of data string, and the minimum storage *reserved* in memory. The INFORMIX-SE engine does not support this data type, but you can declare 4GL VARCHAR variables.

The declared *maximum size* of VARCHAR data types can range from 1 to 255. If you specify no maximum size, the default is 1. You can store shorter character strings than the maximum size, but not longer strings. In a form specification file, you cannot specify size parameters for a FORMONLY VARCHAR field; here the *maximum size* defaults to the field length in the screen layout.

The minimum *reserved size* can range from 0 to 255 bytes, but this cannot be greater than the declared *maximum size*. (Like the *precision* specification in FLOAT or DOUBLE PRECISION data type declarations, **4GL** accepts this value for compatibility with **SQL** syntax, but does not use this value.)

When you assign a value to a VARCHAR variable, only the data characters are stored, but neither **4GL** nor the database engine strips a VARCHAR value of user-entered trailing blanks. Unlike CHAR values, VARCHAR values are *not* padded with blanks to the declared maximum size.

VARCHAR values are compared to CHAR values and to other VARCHAR values in 4GL Boolean expressions in the same way that CHAR values are compared: the shorter value is padded on the right with spaces until the values have equal lengths. Then they are compared for the full length.

NLS

When accessing an NLS database in the standard way (Implicit NLS), all character data in the database is sorted and compared according to the locale established at the time of database creation. For example, the 4GL user cannot create character data columns that sort in US English if the database locale (the locale the database was created in) is non-US-English. Although character columns defined by a 4GL program are defined within the program as type CHAR (or VARCHAR, if variable-length), they are interpreted by the server as type NCHAR (or NVARCHAR) if the database is non-US-English. This process is known as *implicit mapping*.

4GL supports the Implicit and Open NLS environments. Explicit NLS is not supported in **4GL**. For more information, see Appendix E, "Native Language Support Within INFORMIX-4GL."

Data Type Conversion

4GL can assign the value of a number, character string, or point in time to a variable of a different data type. 4GL performs data-type conversion without objection when the process makes sense. If you assign a number expression to a CHAR variable, for example, 4GL converts the resulting number to a string. In an arithmetic expression, 4GL converts the string representation of a number or of a date to an appropriate number or date.

An error is issued only if 4GL cannot make the conversion. For example, it converts the string "`123.456`" to the number `123.456` in an arithmetic expression, but adding the string "John" to a number produces an error.

The global **status** variable is not reset when a conversion error occurs, unless you specify the ANY ERROR keywords (without CONTINUE) in a WHENEVER compiler directive, or include the -ANYERR command-line argument.

Converting from Number to Number

When you pass a value from one number data type to another, the *destination* data type must be able to store all of the *source* value. For example, if you assign an INTEGER value to a SMALLINT variable, the conversion will fail if the absolute value is larger than 32,767. The same situation can occur if you attempt to transfer data from FLOAT or SMALLFLOAT variables or database columns to INTEGER, SMALLINT, or DECIMAL data types.

Page 3-325 lists the kinds of errors that you might encounter when you convert values from one number data type to another. For example, if you convert a FLOAT value to DECIMAL(4,2), 4GL or the database engine rounds off

the floating-point value before storing it as a fixed-point number. This can sometimes result in overflow or rounding error, depending on the declared precision of the DECIMAL data type.

The **SQLAWARN[5]** character of the global **SQLCA** record is set to W after any FLOAT or SMALLFLOAT value is converted to a DECIMAL value.

Converting Numbers in Arithmetic Operations

INFORMIX-4GL performs all arithmetic operations on DECIMAL values, regardless of the declared data types of the operands. The data type of the receiving variable or constant determines the format of the stored or printed result. The following rules apply to the precision and scale of the DECIMAL variable that results from an arithmetic operation on two numbers:

- All operands, if not already DECIMAL, are converted to DECIMAL, and the result of the arithmetic operation is always a DECIMAL:

Source Operand	Converted Operand
FLOAT	DECIMAL(16)
INTEGER	DECIMAL(10,0)
MONEY(*p*)	DECIMAL(*p*,2)
SMALLFLOAT	DECIMAL(8)
SMALLINT	DECIMAL(5,0)

- In addition and subtraction, INFORMIX-4GL adds trailing zeros to the operand with the smaller scale, until the scales are equal.
- If the data type of the result of an arithmetic operation requires the loss of significant digits, then INFORMIX-4GL reports an error.
- Leading or trailing zeros are not considered significant digits, and do not contribute to the determination of precision and scale.
- The precision and scale of the result of an arithmetic operation depend on the precision and scale of the operands and on the type of arithmetic expression. The rules of 4GL are summarized in the chart that follows for arithmetic with operands that have a definite scale. If one operand has no scale (that is, a floating decimal), the result is a floating decimal.

In this chart, p_1 and s_1 represent the precision and scale of the *first* operand, and p_2 and s_2 represent the precision and scale of the *second* operand.

Operation	Precision and Scale of Returned Value	
Addition and Subtraction	Precision: Scale:	$MIN(32, MAX(p_1 - s_1, p_2 - s_2) + MAX(s_1, s_2) + 1)$ $MAX(s_1, s_2)$
Multiplication	Precision: Scale:	$MIN(32, p_1 + p_2)$ $s_1 + s_2$
Division	Precision: Scale:	32 $MAX(0, 32 - p_1 + s_1 - s_2)$

The USING operator can override the default scale when output is displayed.

Converting Between DATE and DATETIME

You can convert DATE values to DATETIME values. If the DATETIME precision includes time units smaller than *day*, however, INFORMIX-4GL either ignores the time-of-day units, or else fills them with zeros, depending on the context. The examples that follow illustrate how these two data types are converted; it is assumed here that the default display format for DATE values is *mm/dd/yyyy*:

- If a DATE value is specified where a DATETIME YEAR TO DAY is expected, 4GL converts the given DATE value to a DATETIME value. For example, `08/15/1993` becomes `1993-08-15`.
- If a DATETIME YEAR TO DAY value is specified where a DATE is expected, then `1993-08-15` becomes `08/15/1993`.
- If a DATE value is specified where a DATETIME YEAR TO FRACTION (or TO SECOND, TO MINUTE, or TO HOUR) is expected, then 4GL converts the DATE value to a DATETIME value, and fills any smaller DATETIME units with zeros. Thus, `08/15/1993` becomes `1993-08-15 00:00:00`.
- If a DATETIME YEAR TO SECOND to DATE, value is specified where a DATE is expected, then 4GL converts the DATETIME value to a DATE value, but drops any units more precise than DAY. Thus, `1993-08-15 12:31:37` becomes `08/15/1993`.

The EXTEND( ) operator can return a DATETIME value from a DATE operand.

Converting CHAR to DATETIME or INTERVAL Data Types

You can specify DATETIME and INTERVAL data in the *literal* forms described in previous sections or as *quoted strings*. The next examples illustrate both formats for December 5, 1974, and for a time interval of nearly 18 days:

```
DEFINE mytime DATETIME YEAR TO DAY,
       myval INTERVAL DAY TO SECOND

LET mytime = DATETIME(74-12-5) YEAR TO DAY
LET mytime = "74-12-5"

LET myval = INTERVAL(17 21:15:30) DAY TO SECOND
LET myval = "17 21:15:30"
```

Values that are specified as suitably formatted character strings are automatically converted into DATETIME or INTERVAL values. You can use character strings whenever you specify information for all time units declared for that DATETIME or INTERVAL variable or named constant.

When a character string is converted into a DATETIME or INTERVAL value, **4GL** assumes that the character string includes information about all the declared time units. You cannot use character strings to enter DATETIME or INTERVAL values for a subset of time units, because this produces ambiguous values. If the character string does not contain information for all time units, **4GL** returns an error, as in these examples:

```
DEFINE tyme DATETIME YEAR TO DAY,
       mynt INTERVAL DAY TO SECOND

LET tyme = DATETIME(5-12) MONTH TO DAY     --Valid
LET tyme = "5-12"                          --Error!

LET mynt = INTERVAL(11:15) HOUR TO MINUTE   --Valid
LET mynt = "11:15"                          --Error!
```

The previous DATETIME example (variable **tyme**) assigns a MONTH TO DAY value to a variable declared as YEAR TO DAY. Entering only these values is valid in the first LET statement, because the qualifier of the DATETIME literal specifies that there is no *year* data. In this case, **4GL** automatically supplies the value of the current year. The character string, however, does not indicate what information is omitted; **4GL** does not know whether the “5-12” refers to *year* and *month*, or *month* and *day*, so it returns an error.

The previous INTERVAL example (variable **mynt**) assigns an HOUR TO MINUTE value to a variable declared as DAY TO SECOND. The first LET statement simply pads the value with zeros for *day* and *second*. The second LET statement produces a conversion error, however, since **4GL** does not know whether the "`11:15`" specification means HOUR TO MINUTE or MINUTE TO SECOND.

Converting Between Number and Character Data Types

You can store a CHAR or VARCHAR value in a number variable and *vice versa*. But if the CHAR or VARCHAR value contains any characters that are not valid in a number data type (for example, the letters `l` *or* `O` instead of the digits `1` or `0`), then INFORMIX-4GL returns a data type conversion error.

NLS

If NLS is active and you convert numeric or monetary values by using the LET statement, the conversion process inserts locale-specific separators and currency symbols into the created strings, not US English separators and currency symbols. Monetary values take on separators and currency symbols specified by LC_MONETARY. Numeric values take on separators specified by LC_NUMERIC. This happens regardless of you including a USING clause in the LET statement. However, if DBFORMAT or DBMONEY is set, these settings override settings in LC_ variables.

Converting Large Binary Data Types

You can store a TEXT value in a BYTE data type. Space permitting, you can also store all or part of a TEXT value in a CHAR or VARCHAR variable. No other data type conversions involving large binary data types are supported directly by INFORMIX-4GL.

Summary of Compatible 4GL Data Types

The following table shows which pairs of 4GL data types are *compatible*.

- Unshaded cells show the types of values (listed in the top row) that 4GL can assign to each type of variable (listed in the left-hand column).
- Shaded cells indicate *incompatible* pairs of data types, for which INFORMIX-4GL does not support automatic data type conversion.

Value to be Passed: / Receiving Data Type	CHAR	VARCHAR	INTEGER	SMALLINT	FLOAT	SMALLFLOAT	DECIMAL	MONEY	DATE	DATETIME	INTERVAL
CHAR	①	①	①	①	①	①	①	①❶	①❷	①	①
VARCHAR	①	①	①	①	①	①	①	①❶	①❷	①	①
INTEGER	②③	②③			③④	③④	③④	③④	❹		
SMALLINT	②③	②③	③		③④	③④	③④	③④	③❹		
FLOAT	②③⑤	②③⑤	❸	❸			③	③	❹		
SMALLFLOAT	②③⑤	②③⑤	❸⑤	❸	⑤		③⑤	③⑤	❹⑤		
DECIMAL	②③⑥	②③⑥	③	③	③⑥	③⑥	③⑥	③⑥	③❹		
MONEY	②③⑥	②③⑥	③	③	③⑥	③⑥	③⑥	③⑥	③❹		
DATE	②	②	❹	❹	③④❹	③④❹	③④❹	③④❹		❺❼	
DATETIME	②	②							❻❼	⑦❼	
INTERVAL	②	②									③⑦

Symbols in cells refer to notes on the next page. These apply when the data types of the passed value and of the receiving variable are not identical:

- Light circles (①) indicate restrictions that can result in conversion failure, or in discrepancies between the passed value and the receiving variable.
- Dark circles (❶) indicate features that typically do not cause conversion errors, but that may produce unexpected data formats or values.

This table also applies to simple members of RECORD variables and to simple elements of ARRAY variables. INFORMIX-4GL does not support automatic data-type conversion of values of the BYTE or TEXT data types.

Notes on Automatic Data Type Conversion

In the previous table, numbers within light circles (①) indicate restrictions that can cause the data type conversion to fail, or that can sometimes result in discrepancies between the passed value and the receiving variable:

① If the result of converting a value to a character string is longer than the receiving variable, then the character string is truncated from the right.

② Character string values must depict a literal of the receiving data type.

③ If the value exceeds the range of the receiving data type, then an overflow error occurs.

④ Any fraction is truncated.

⑤ If the passed value contains more significant digits than the receiving data type supports, then low-order digits are discarded.

⑥ If the passed value contains more fractional digits than the receiving data type supports, then low-order digits are discarded.

⑦ Differences between qualifiers may cause truncation from the left or right.

Numbers in dark circles (❶) indicate less critical data-type conversion features. These are not associated with error messages, but they can result in the assignment of unexpected values, or of values with unexpected formats:

❶ DBMONEY controls the format of the converted string.

❷ DBDATE controls the format of the converted string.

❸ Rounding error may produce an assigned value with a fractional part.

❹ An integer value corresponding to a count of *days* is assigned.

❺ An implicit EXTEND (*value*, YEAR TO DAY) is performed by 4GL.

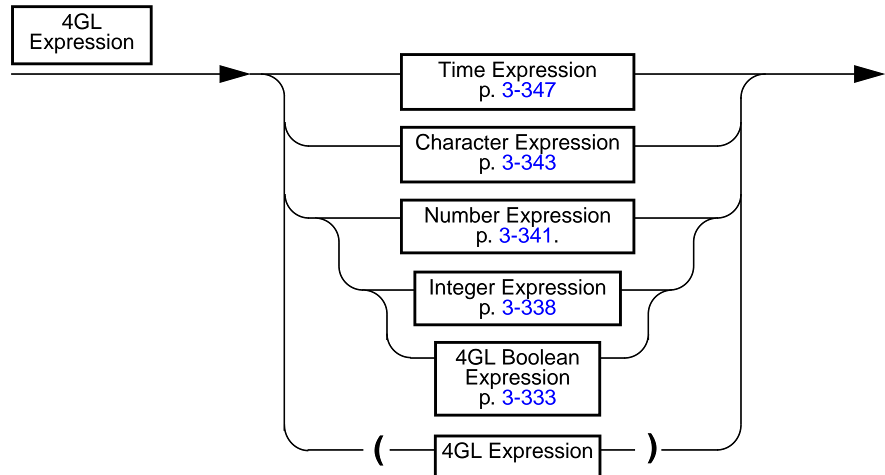
❻ The DATE becomes a DATETIME YEAR TO DAY literal before assignment.

❼ If the passed value has less precision than the receiving variable, then any missing time unit values are obtained from the system clock.

You may wish to avoid writing code that applies automatic conversion to DATETIME variables declared with time units smaller than YEAR as the first keyword of the DATETIME qualifier (page 3-349), unless default values that feature (❼) assigns from the system clock are useful in your application.

Expressions of 4GL

A 4GL *expression* is a sequence of operands, operators, and parentheses that INFORMIX-4GL can evaluate as a single value.



Usage

Statements, functions, form specifications, operators, and expressions can have expressions as arguments, components, or operands. The context where an expression appears, as well as its syntax, determines the *data type* of its returned value. It is convenient to classify 4GL expressions into the following five categories, based on the data type of what they return:

Boolean a value that is either TRUE or FALSE or NULL

Integer a whole-number value of data type INT or SMALLINT

Number a value of any *number* data type, as listed on page 3-295

Character a character string of data type CHAR or VARCHAR

Time a value of data type DATE, DATETIME, or INTERVAL

In this manual, if the term *"4GL expression"* is not qualified as one of these five specific data types, then the expression can be any of these data types.

As the diagram suggests, 4GL Boolean expressions are special cases of integer expressions, and integer expressions are a logical subset of number expressions. You can substitute a 4GL Boolean or integer expression where a number expression is valid (unless this results in an attempt to divide by zero).

The topics that are discussed in this section include the following:

Topic	Page
Components of 4GL Expressions	3-327
4GL Boolean Expressionss	3-333
Integer Expressions	3-338
Number Expressions	3-341
Character Expressions	3-343
Time Expressions	3-347

Components of 4GL Expressions

An expression of 4GL can include the following components:

- Operators, as listed on the next page (and on page 4-10)
- Operands, including the following:
  - Other 4GL expressions (page 3-326)
  - Named values (page 3-331)
  - Function calls returning a single value (page 3-332)
  - Literal constants (pages 3-340, 3-342, 3-343, 3-349, 3-351, and 3-355)
  - Field names (3-328, 3-359)
- Parentheses, to override the default precedence of operators (page 3-328).

Parentheses in 4GL Expressions

You can use parentheses (( )) as you would in algebra to override the default order of precedence (page 3-328) of 4GL operators. In mathematics, this use of parentheses represents the "associative" operator. It is, however, a convention in computer languages to regard this use of parentheses symbols as *delimiters*, rather than as operators. (Do not confuse this use of parentheses to specify *operator precedence* with parentheses to enclose arguments in *function calls*, or to delimit other lists.)

Operators in 4GL Expressions

The *operators* listed on the next page can appear in 4GL expressions. Expressions with several operators are evaluated according to precedence, from highest (13) to lowest (1), as indicated in the left-hand (**P**) column. The fourth column (**A**) indicates the associativity, if any, of each operator. See the page references in the right-hand columns of the following tables for additional information about individual operators of INFORMIX-4GL.

Precedence (P) and Associativity (A) of 4GL Operators

P	Operator	Description	A	Example	Page
13	.	record membership	left	**myrec.memb**	3-313
	[ ]	array index *or* substring	left	**ar[i, 6, k][2,(*integer-expr*)]**	3-344
	( )	function call	none	**myfun(var1**, *expr***)**	3-332
12	UNITS	single-qualifier interval	left	**(***integer-expr***) UNITS DAYS**	3-347
11	+	unary plus	right	+ **(***number-expr***)**	3-341,
	-	unary minus	right	- **numbarray_var3[i, j, k]**	3-341
10	* *	exponentiation (by integer)	left	**(***number-expr***)** ** **(***integer-expr***)**	3-320,
	MOD	modulus (of integer)	left	**(***integer-expr***) MOD (***integer-expr***)**	3-339
9	*	multiplication	left	**x** * **(***number-expr***)**	3-320,
	/	division	left	**(***number-expr***) / arr[y]**	3-339
8	+	addition	left	**(***number-expr***)** + **(***number-expr***)**	3-320,
	-	subtraction	left	**(x - y) - (***number-expr***)**	3-339
7	LIKE	string comparison	right	**(***character-expr***) LIKE "%z_%"**	3-335
	MATCHES	string comparison	right	**(***character-expr***) MATCHES "*z?"**	3-335
6	<	*test for*: less than	left	**(***expr1***)** < **(***expr2***)**	3-334
	<=	less than *or* equal to	left	x <= **yourfun**(y,z)	3-334
	= *or* ==	equal to	left	**x** = *expr*	3-334
	>=	greater than *or* equal to	left	x >= **FALSE**	3-334
	>	greater than	left	**var1** > *expr*	3-334
	!= *or* <>	not equal to	left	**myrec.memb<>length(var1)**	3-334
5	IS NULL	*test for*: NULL	left	**x IS NULL**	3-336
	IS NOT NULL	*test for*: NULL	left	*expr* **IS NOT NULL**	3-336
4	NOT	logical inverse	left	**NOT ((***expr***) IN (y,DATE))**	3-333
3	AND	logical union	left	*expr1* **AND fun(***expr2***,-y)**	3-333
2	OR	logical alternation	left	**LENGTH(***expr1***,j) OR** *expr2*	3-333
1	ASCII	return ASCII character	right	**LET x** = ASCII **(***integer-expr***)**	3-344
	CLIPPED	delete trailing blanks	right	DISPLAY **poodle CLIPPED**	3-344
	COLUMN	begin character display	right	DISPLAY **COLUMN 58, "30"**	3-344
	SPACES	insert blank spaces	right	PRINT **(***integer-expr***) SPACES**	3-344
	USING	format character string	right	**TODAY USING "yy/mm/dd"**	3-344
	WORDWRAP	multiple line text display	right	PRINT **odyssey WORDWRAP**	3-344

P values are ordinal numbers that may change if future releases add new operators. Also of lowest precedence (**P** = 1) are these built-in operators:

- The *field* operators FIELD_TOUCHED( ), GET_FLDBUF( ), and INFIELD( )
- The built-in *report* operators SPACE, LINENO, and PAGENO

- The built-in *time* operators CURRENT, DATE, DATE( ), DAY( ), EXTEND( ), MDY( ), MONTH( ), TIME, TODAY, WEEKDAY( ), and YEAR( )

See the following table and Chapter 4 for more about the operators of 4GL.

Data Types of Operands and of Returned Values

Expression	Left (= x)	Right (= y)	Returned Value	Page
x . y	**RECORD**	Any	Same as **y**	3-313
w [x , y]	**INT** *or* **SMALLINT**	**INT** *or* **SMALLINT**	Any *or* Character	3-344
(y)		Any	Any	3-332
x UNITS	**INT** *or* **SMALLINT**		**INTERVAL**	4-89
+ **y**		Number *or* **INTERVAL**	Same as **y**	3-341,
- **y**		Number *or* **INTERVAL**	Same as **y**	3-347
x * * **y**	Number	**INT** *or* **SMALLINT**	Number	3-341,
x MOD **y**	**INT** *or* **SMALLINT**	**INT** *or* **SMALLINT**	**INT** *or* **SMALLINT**	3-356
x * **y**	Number *or* **INTERVAL**	Number	Number *or* **INTERVAL**	3-341,
x / **y**	Number *or* **INTERVAL**	Number	Number *or* **INTERVAL**	3-356
x + **y**	Number *or* Time	Number *or* Time	Number *or* Time	3-341,
x - **y**	Number *or* Time	Number *or* Time	Number *or* Time	3-356
x LIKE **y**	Character	Character	Boolean	4-33
x MATCHES **y**	Character	Character	Boolean	4-33
x < **y**	Any simple data type	Same as **x**	Boolean	3-358
x <= **y**	Any simple data type	Same as **x**	Boolean	3-358
x = **y** *or* **x** == **y**	Any simple data type	Same as **x**	Boolean	3-358
x >= **y**	Any simple data type	Same as **x**	Boolean	3-358
x > **y**	Any simple data type	Same as **x**	Boolean	3-358
x != **y** *or* **x** <> **y**	Any simple data type	Same as **x**	Boolean	3-358
x IS NULL	Any		Boolean	4-33
x IS NOT NULL	Any		Boolean	4-33
NOT **y**		Boolean	Boolean	4-31
x AND **y**	Boolean	Boolean	Boolean	4-31
x OR **y**	Boolean	Boolean	Boolean	4-31
ASCII **y**		**INT** *or* **SMALLINT**	Character	4-28
x CLIPPED	Character		Character	4-38
COLUMN **y**		**INT** *or* **SMALLINT**	Character	4-40
x SPACES	**INT** *or* **SMALLINT**		Character	4-82
x USING **"y"**	Char.,**DATE**,**MONEY**	Character	Character	4-91
x WORDWRAP	Character		Character	4-102

Where no data type is listed, the operator has no left (or else right) operand. If an operand is not of the data type(s) listed here for a given operator, then 4GL attempts data type conversion, as summarized on page 3-324.

Most 4GL operators do not support RECORD nor ARRAY operands, but do accept a simple variable operand that is an array element or a record member.

Differences Between 4GL and SQL Expressions

Expressions in SQL statements (and in SPL statements) are evaluated by the database engine, not 4GL. The set of operators that can appear in SQL or SPL expressions resembles the set of 4GL operators, but they are not identical.

A 4GL program can include SQL operators, but some are restricted to SQL statements. Similarly, some SQL and SPL operands are not valid in 4GL expressions. The SQL identifiers of *databases*, *tables*, or *columns* can appear in a LIKE clause or field name (page 3-359) in 4GL statements, but these SQL and SPL operands and operators cannot appear in other 4GL expressions:

- SQL identifiers, such as column names
- The names of SPL variables
- The SQL keywords USER and ROWID
- Built-in or aggregate SQL functions that are not part of 4GL
- The || string concatenation operator
- The BETWEEN ... AND and IN operators (except in form specification files)
- The EXISTS, ALL, ANY, or SOME keywords of SQL Boolean expressions

Conversely, you *cannot* include the following INFORMIX-4GL operators in SQL or SPL expressions:

- The member-selection (.) operator for records
- The arithmetic operators for exponentiation (**) and modulus (MOD)
- The string operators ASCII, COLUMN, SPACE, SPACES, and WORDWRAP
- The field operators FIELD_TOUCHED( ), GET_FLDBUF( ), and INFIELD( )
- The report operators LINENO and PAGENO
- The time operators DATE and TIME

See the *Informix Guide to SQL: Reference* for the valid syntax of SQL expressions and of SPL expressions.

Operands in 4GL Expressions

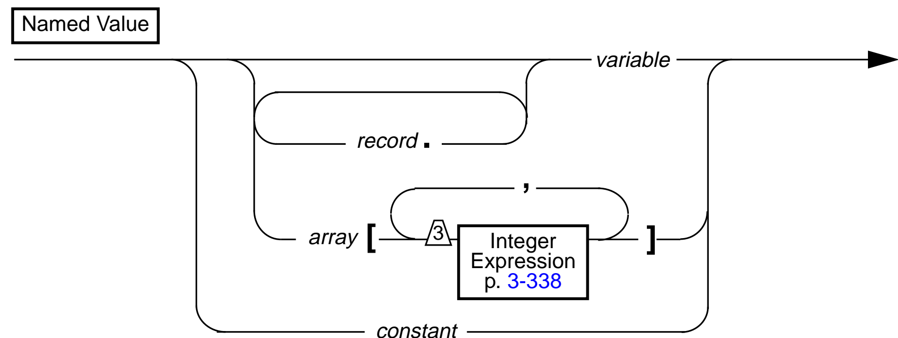
Operands of 4GL expressions can be any of the following:

- Other 4GL *expressions* (page 3-326)
- *Named values* (page 3-331)
- *Function calls* that return one value (page 3-332)
- *Literal constants* (pages 3-340, 3-342, 3-343, 3-349, 3-351, and 3-355)

Sections that follow describe these operands of 4GL expressions.

Named Values as Operands

A 4GL expression can include the name of a variable of any simple data type (as identified on page 3-294) or the constants TRUE or FALSE. The variable can also be a simple member of a record, or a simple element of an array:



array	is the name of a structured variable of the ARRAY data type. The comma-separated *expression* list specifies the index of an element within the declared size of the array.
constant	is the name of the built-in constant TRUE or FALSE.
record	is the name of a structured variable of the RECORD data type.
variable	is the name of a 4GL program variable.

In two special cases, other identifiers can be operands in 4GL expressions:

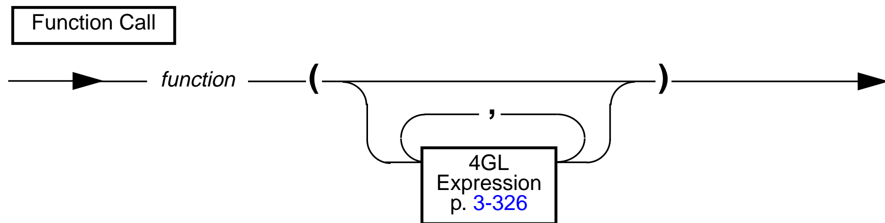
- Conditional COLOR attributes in form specification files can use a *field tag* where a named value is valid in the syntax of a 4GL Boolean expression.
- The built-in FIELD_TOUCHED( ), GET_FLDBUF( ), and INFIELD( ) operators can take *field names* (page 3-359) as operands. See Chapter 4 for the syntax of these field operators.

If the variable is a member of a record, qualify it with the record name prefix, separated by the period (.) symbol as the record membership operator.

Variables of the BYTE or TEXT data types cannot appear in expressions, except as operands of the IS NULL or IS NOT NULL operators (pages 3-336 and 4-33) or (for TEXT variables only) the WORDWRAP operator (pages 4-102 and 6-50).

Function Calls as Operands

A 4GL expression can include calls to functions that return exactly one value:



function is the name of a function. The parentheses are required, regardless of whether the function takes any arguments.

The *function* can be a programmer-defined or built-in function, provided that it returns a single value of a data type that is valid in the expression. (Function calls as *arguments* can return multiple values.)

See the FUNCTION statement (page 3-111) or Chapter 4 for more information about declaring, defining, and invoking INFORMIX-4GL functions.

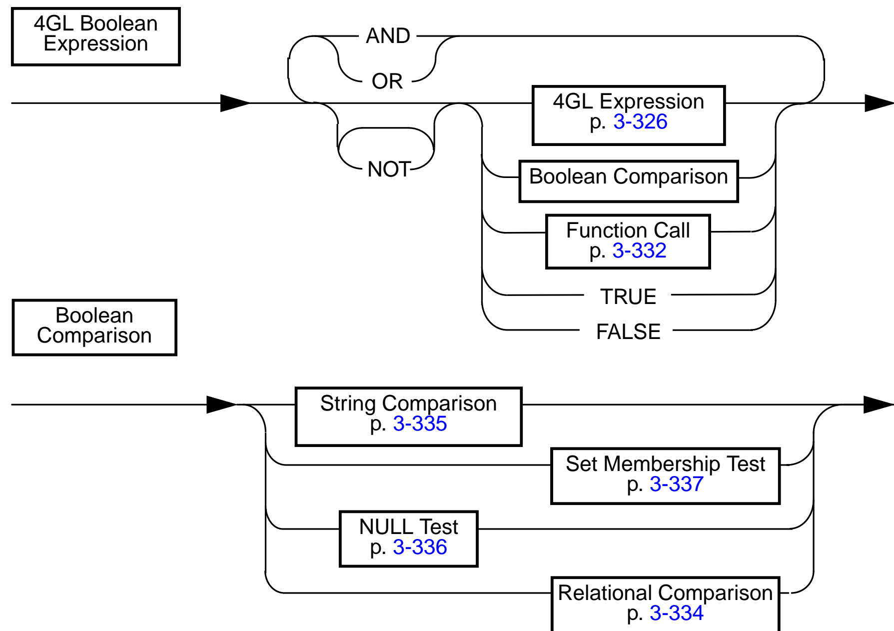
Expressions as Operands

Two expressions cannot appear consecutively without some separator, but you can nest expressions within expressions. In any context, however, the complexity of a 4GL expression is restricted. If an error message indicates that an expression is too complex, you should substitute two or more simpler expressions that 4GL can evaluate, and then combine these values.

If an expression returns a different data type from what 4GL expects in the context, 4GL attempts data type conversion, as summarized on page 3-324.

4GL Boolean Expressions

A 4GL *Boolean expression* is one that returns either TRUE (defined as `1`) or FALSE (defined as `0`) or NULL. The syntax of Boolean expressions in 4GL statements is not identical to that of *Boolean conditions* in SQL statements. Boolean expressions of INFORMIX-4GL have the following syntax:



Any type of expression can be a 4GL Boolean expression. You can use an INT or SMALLINT variable to store the returned value.

Logical Operators

The *logical operators* AND, OR, and NOT combine Boolean values into a single 4GL Boolean expression. AND, OR, and NOT produce the following results (where the symbol `T` means TRUE, `F` means FALSE, and `?` means NULL):

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

If one or both operands of a logical operator have NULL values, the result can in some cases also be NULL. For example, if **var1** = 0 and **var2** = NULL, then

```
LET x = var1 OR var2
```

assigns to the variable **x** a NULL value. The NOT operator is recursive.

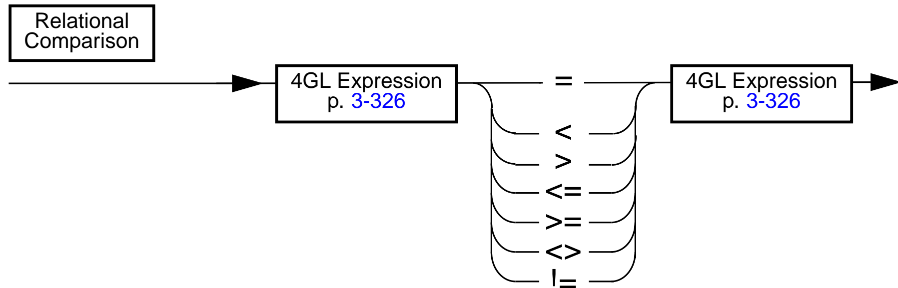
Boolean Comparisons

Boolean comparisons use relational operators to test any type of expression for equality or inequality. You can also use the IS NULL operator to test for NULL values, or the MATCHES or LIKE operator to compare *character strings*.

Boolean expressions in the CASE, IF, or WHILE statements, or in the WHERE clause of a COLOR attribute specification return FALSE if any element of the comparison is NULL, unless it is the operand of the IS NULL operator.

Relational Comparisons

This is the syntax for relational comparisons in 4GL Boolean expressions:



Boolean expressions in 4GL statements can use these *relational operators* (=, ==, <, >, <=, >=, <>, or !=, as defined on page 3-328) to compare operands. For example, each of these comparisons evaluates to TRUE or FALSE:

Expression	Value
(2+5)* 3 = 18	FALSE
14 <= 16	TRUE
"James" = "Jones"	FALSE

Use a NULL test (page 3-337) if you want to detect and exclude NULL values from Boolean comparisons. In this CASE statement fragment, the value of the comparison is NULL if the value of **salary** or of **last_raise** is NULL:

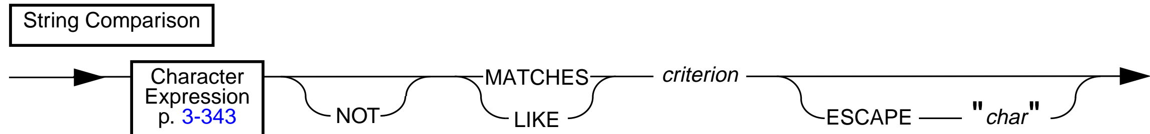
```
WHEN salary * last_raise < 25000
```

The LIKE and MATCHES Operators

In *character string* comparisons, you can use the LIKE or MATCHES operators to test whether a character value matches a quoted string that can include *wildcard characters*. If either operand evaluates to NULL, then the entire string comparison evaluates to NULL. Use a NULL test (page 3-337) if you want to detect and exclude NULL values.

You can use the following syntax to compare character strings:

String Comparison



char is a single ASCII character, enclosed between a pair of single (') or double (") quotation marks, to specify an escape symbol.

criterion is a character expression (page 3-343). The string that it returns can include literal characters, wildcards, and other symbols.

MATCHES and LIKE support different wildcards. If you use MATCHES, you can include the following wildcard characters in the right-hand operand:

Symbol	Effect
*	An asterisk (*) matches a string of any length (including zero characters).
?	A question mark (?) matches any single character.
[ ]	Square brackets ([]) match any of the enclosed characters.
-	A hyphen (-) between characters in brackets means a range in the ASCII collating sequence. For example, `[a-z]` matches any lowercase letter.
^	A caret (^) as the first character in the brackets matches any character that is not listed. For example, `[^abc]` matches any character except `a`, `b`, or `c`.
\	A backslash (\ ) causes **4GL** to treat the next character as a literal character, even if it is one of the special symbols in this list. For example, you can match * or ? by * or \? in the string.

The following WHERE clause tests the contents of character field **field007** for the string `ten`. Here the `*` wildcards specify that the comparison is TRUE if `ten` is found alone or in a longer string, such as `often` or `tennis shoe`:

```
COLOR = RED WHERE field007 MATCHES "*ten*"
```

If you use the operator LIKE to compare strings, then the wildcard symbols of MATCHES have no special significance, but you can use the following wildcard characters of LIKE within the right-hand quoted string:

Symbol	Effect
%	A percent sign (%) matches zero or more characters.
_	An underscore (_) matches any single character.
\	A backslash (\ ) causes 4GL to treat the next character as a literal (so you can match % or _ by \% or _).

The next example tests for the string `ten` in the character variable **string**, either alone or in a longer string:

```
IF string LIKE "%ten%"
```

The next example tests whether a substring of a character variable (or else an element of a two-dimensional array) contains an underscore symbol. The backslash is necessary, because underscore is a wildcard symbol with LIKE.

```
IF horray[3,8] LIKE "%\_%" WHERE >> out.a
```

You can replace backslash as the literal symbol. If you include an ESCAPE *char* clause in a LIKE or MATCHES specification, then INFORMIX-4GL interprets the next character that follows *char* as a literal in the preceding character expression, even if that character corresponds to a special symbol of the LIKE or MATCHES operator. The double quote (**"**) symbol cannot be *char*.

For example, if you specify ESCAPE `z`, the characters `z_` and `z?` in a string represent the literal character _ and `?`, rather than wildcards. Similarly, characters `z%` and `z*` represent the characters **%** and *****. Finally, the characters `zz` in the string represent the single character `z`. The following expression is TRUE if the variable **company** does not include the underscore character:

```
NOT company LIKE "%z_%" ESCAPE "z"
```

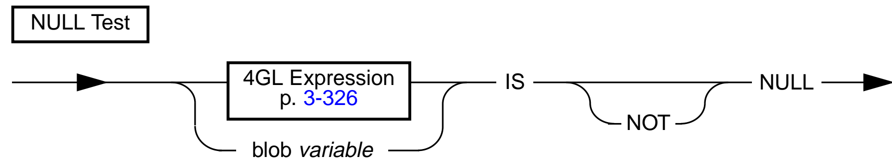
NLS

If you are using NLS, the evaluation of logical comparisons and MATCHES, LIKE, and BETWEEN expressions containing character arguments is dependent on LANG and LC_COLLATE settings in an NLS database.

The NULL Test

If any operand of a 4GL Boolean comparison or is NULL, then the value of the comparison is FALSE (rather than NULL), unless the IS NULL keywords are also included in the expression. Applying the NOT or IS NOT NULL operator to a NULL operand does not change its FALSE evaluation.

If you need to process NULL values in a different way from other values, you can use the IS NULL operator to test for NULL value. It has this syntax:



BLOB variable is the name of a variable of the BYTE or TEXT data type.

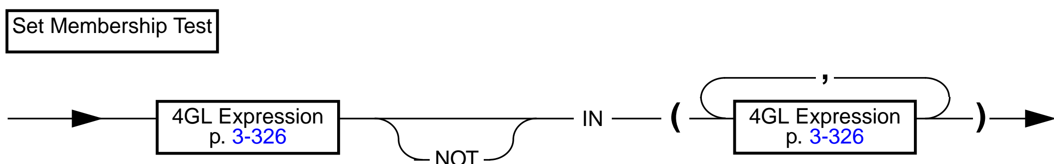
Without the NOT keyword, this comparison returns TRUE if the operand has a NULL value. Otherwise, it returns FALSE.

If you also include the NOT keyword after IS, this comparison returns FALSE if the operand has a NULL value. Otherwise, it returns TRUE.

Note: *The NULL test (like the WORDWRAP string operator with TEXT variables) is an exception to the general rule that variables of the BYTE or TEXT data types cannot appear in* 4GL *expressions.*

Set Membership Test

The IN( ) operator cannot appear in any 4GL statements that is not also an SQL statement, but it can appear in the COLOR attribute specification of a 4GL form. This is the syntax of the IN( ) operator to test for set membership:



If you omit the NOT operator, this test returns TRUE if any expression in the comma-separated list at the right matches the expression on the left.

If you include the NOT operator, the test returns FALSE if no expression in the list matches the expression on the left.

The list of members must be in parentheses. (Like a DATETIME or INTERVAL literal, the IN( ) operator requires parentheses but is *not* a function call.)

Data Type Compatibility

You may get unexpected results if you use relational operators with expressions of dissimilar data types. In general, you can compare numbers with numbers, character strings with strings, and time values with time values.

If a *time expression* operand of a 4GL Boolean expression is of the INTERVAL data type, then any other time expression that is compared to it by a relational operator must also be an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value). See the section "Data Type Conversion" on page 3-319 for additional information about data type compatibility in expressions.

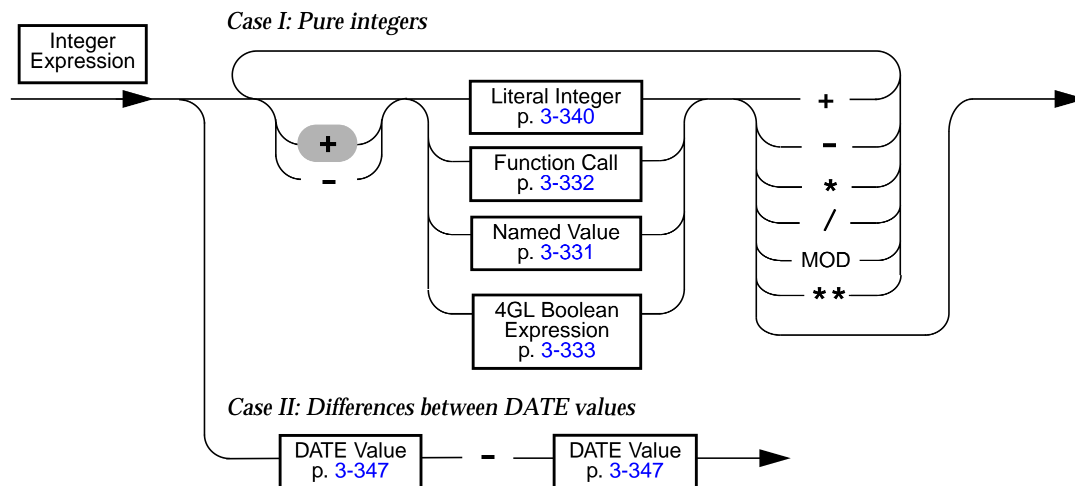
Evaluating 4GL Boolean Expressions

In contexts where a 4GL Boolean expression is expected, INFORMIX-4GL applies the following rules after it evaluates the expression:

- If the value is a non-zero real number (or a character string representing a non-zero number) or a non-zero INTERVAL, or any DATE or DATETIME value, or a TRUE value returned by a Boolean function like INFIELD( ), or the integer constant TRUE, then the 4GL Boolean value is TRUE.
- If the value is NULL, but the expression is the operand of the IS NULL keywords, then the value of the 4GL Boolean expression is TRUE.
- If the value is NULL, and the expression is not an operand of a NULL test, nor an element in any Boolean comparison (page 3-334) or conditional statement of 4GL (IF, CASE, WHILE), then the expression returns NULL.
- Otherwise, the 4GL Boolean expression is evaluated as FALSE.

Integer Expressions

An *integer expression* returns a whole number. It has this syntax:



Here any *function call* or *named value* returns an integer. Logical restrictions on using DATE values as integer expressions are discussed on page 3-356.

Integer expressions can be components of expressions of every other type. Like Boolean expressions (page 3-333), integer expressions are a logical subset of *number* expressions (page 3-341), but they are separately described here because some 4GL operators, statements, form specifications, operators, and built-in functions are restricted to integer values, or to positive integers.

Binary Arithmetic Operators

Six binary arithmetic operators can appear in an integer expression, and can take integer expressions as both the right-hand and left-hand operands:

Operator Symbol	Operator Name	Name of Result	Precedence
**	exponentiation	power	12
mod	modulus	integer remainder	12
*	multiplication	product	11
/	division	quotient	11
+	addition	sum	10
-	subtraction	difference	10

Note: *All arithmetic calculations are performed after converting both operands to DECIMAL values (but MOD operands are first converted to INTEGER).*

If an expression has several operators of the same precedence, 4GL processes them from left to right. See page 3-328 for the complete "Precedence" scale for INFORMIX-4GL operators. If any operand of an arithmetic expression is a NULL value, then the entire expression returns NULL.

An *integer expression* specifying an array element or the right-hand MOD operand cannot include exponentiation (**) nor modulus (MOD) operators, and cannot be zero. The right-hand *integer expression* operand of the exponentiation (**) operator cannot be negative. You cannot use "**mod**" as a 4GL identifier.

If both operands of the division (/) operator have INT or SMALLINT data types, 4GL discards any fractional portion of the quotient. An error occurs if the right-hand operand of the division operator evaluates to zero. With some restrictions, 4GL also supports these binary arithmetic operators in number expressions (page 3-341) and in some time expressions (page 3-347).

Differences between two DATE values (page 3-338) are integer expressions. To convert these to type INTERVAL, apply the UNITS DAY operator explicitly.

If you include an 4GL Boolean expression in a context where 4GL expects a number, the Boolean expression is evaluated, and then converted to an integer by the rules: TRUE = 1 and FALSE = 0. An error results if you attempt to divide by zero.

Unary Arithmetic Operators

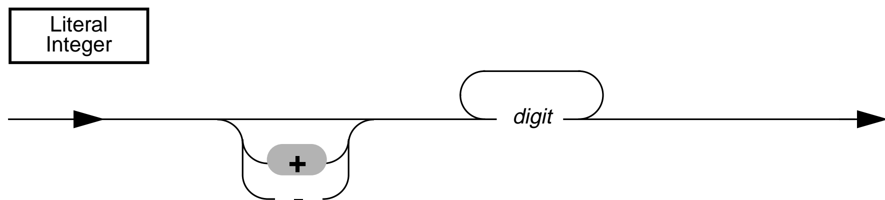
You can use plus (+) and minus (**-**) symbols at the left are *unary operators* to indicate the sign of the expression, or the sign of a component number. For unsigned values, the default is positive (+). Use parentheses (**( )**) to separate the subtraction operator (**-**) from any immediately following unary minus sign, as in "**minuend -(-subtrahend),"** unless you want 4GL to interpret the "**--**" symbols as a comment indicator.

The same rules apply to plus (+) and minus (–) unary operators used with *number* expressions, and with *time* expressions that return INTERVAL values.

The unary plus (+) and minus (–) operators are recursive.

Literal Integers

You must write *literal integers* in base-10 notation, without embedded blank spaces or commas, and without a decimal point:



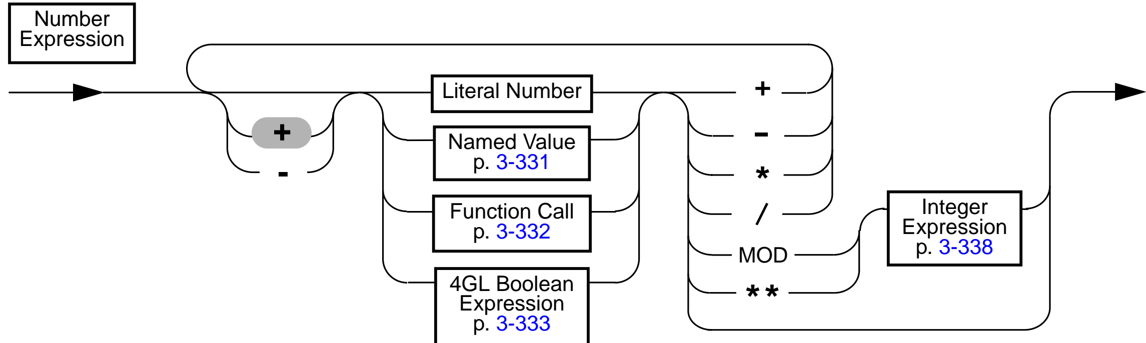
digit is any of the symbols `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, or `0`.

You can precede the integer with unary minus (–) or plus (+) signs:

```
15        -12        13938       +4
```

Number Expressions

A *number expression* is a specification that evaluates to a real number.



Here the function call or named value must return a real number of data type DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, or SMALLINT.

If any operand of an arithmetic operator in a number expression is a NULL value, then 4GL evaluates the entire expression as a NULL value. The range of values in a number expression is that of the receiving data type.

Arithmetic Operators

The sections "Binary Arithmetic Operators" on page 3-339 and "Unary Arithmetic Operators" on page 3-340 apply to number expressions. 4GL converts any modulus (MOD) operand or right-hand operand of the exponentiation (**) operator to INTEGER before conversion to DECIMAL for evaluation; this feature has the effect of discarding any fractional part of the operands.

If both operands are INTEGER, SMALLINT or DATE data types, then the result of any arithmetic operation (including division) is a whole number. If either operand is of data type DECIMAL, FLOAT, MONEY, or SMALLFLOAT, then the returned value may include a fractional part, except in MOD operations.

Literal Numbers

A *literal number* is the base-10 representation of a real number, written as an integer, as a fixed-point decimal number, or in exponential notation.

Literal Number

digit is any of the symbols `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, or `0`.

This cannot include any comma (**,**) nor blank (= ASCII 32) character. The unary plus or a minus sign can precede a literal number or exponent.

There are three kinds of literal numbers:

- *Integer literals* (page 3-340) can exactly represent INTEGER and SMALLINT values. Literal integers have no decimal points, as in these example:

```
10                  -27                 25567
```

- *Fixed-point decimal literals* can exactly represent DECIMAL(*p,s*) and MONEY values. These can include a decimal point. Some examples follow:

```
123.456             .00123456           -123456.0
```

- *Floating-point literals* can exactly represent FLOAT, SMALLFLOAT, and DECIMAL(*p*) values. These literals contain a decimal point and/or exponential notation. These are examples of floating-point literals:

```
123.456e4           -1.23456e2          -123456.0e-3
```

When you use a literal number to represent a MONEY value, do not precede it with a currency symbol. Currency symbols are displayed by 4GL when MONEY values appear in a form or in a report, using whatever the DBMONEY or DBFORMAT environment variable specifies, or else the default **$** symbol.

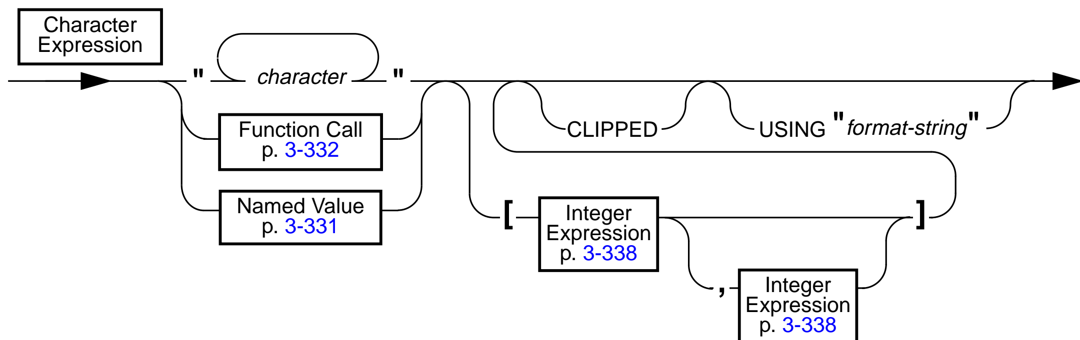
4GL automatically attempts data-type conversion when a literal number is in a different format from the expected data type. If you include a character value in a context that requires a number expression, 4GL attempts to convert the string to a number. (Page 3-324 summarizes data type conversion.)

You may get unexpected results, however, if a literal number in an 4GL Boolean expression is not in a format that can exactly represent the data type of another value with which it is compared by a relational operator. Because of rounding error, for example, relational operators generally cannot return TRUE if one operand returns a FLOAT value, and the other an INTEGER.

Similarly, you will get unpredictable (but probably useless) results if you use literal binary, hexadecimal, or other numbers that are not base-10 where 4GL expects a number expression. You must convert such numbers to a base-10 format before you can use them in a number expression.

Character Expressions

A *character expression* is a specification that evaluates to a character string.



character is one or more characters enclosed between two single (') or double (") quotation marks. (This is sometimes called a "*character string,*" a "*quoted string,*" or a "*string literal.*")

format-string is a quoted string of symbols to specify how 4GL displays the returned character value. (See page 4-91 for details.)

Here the *function call* or *named value* returns a CHAR or VARCHAR value. No variable in a character expression can be of the TEXT data type, except in a NULL test (page 3-336), or as a WORDWRAP operand in a PRINT statement of a 4GL report. As in any 4GL statement or expression, you cannot reference a named value outside its scope of reference. (See page 2-11.)

If a character expression includes a 4GL variable or function whose value is neither of type CHAR nor VARCHAR, 4GL attempts to convert the value to a *character string*. For example, the following program fragment

```
VARIABLE I INTEGER,
    J, K  CHAR(5)
LET I = 4*8
LET J = "FAX"
LET K =  J CLIPPED,I
```

stores the character string "FAX32" in the CHAR variable **K**.

The maximum length of a string value is the same as for the declared data type: up to 32,767 characters for CHAR values, and up to 255 for VARCHAR. (Some 4GL features cannot be applied to strings longer than 512 characters.)

If character expressions are operands of a relational operator (page 3-334), 4GL evaluates both character expressions, and then compares the returned values according to their position within the ASCII collating sequence.

Arrays and Substrings

Any *integer expressions* in brackets that follow the name of an *array* must evaluate to a positive number within a range from 1 to the declared size of the array. For example, **SQLCA.SQLCAWARN[6]** specifies the sixth element of character array **SQLCAWARN** within the **SQLCA** global record.

The pair of *integer expressions* that can follow a character expression specify a *substring*. The first value cannot be larger than the second. Both must be positive, and no larger than the string length (or the receiving data type). For example, **name[1,4]** specifies the first four characters of a program variable called **name**.

Neither the exponentiation (**) nor modulus (MOD) operators can appear in an *integer expression* that specifies an array element or a substring, but parentheses (( )) and the other arithmetic operators (+, -, *, /) are permitted.

String Operators

You can use the USING keyword, followed by a *format string*, to impose a specific format on the character string to which an expression evaluates, or upon any components of a concatenated character expression. (4GL forms and reports support additional features for formatting character values.)

To discard trailing blanks from a character value, apply the CLIPPED operator to the expression, or to any components of a concatenated character expression. See also the WORDWRAP field attribute in forms (page 5-57), and the WORDWRAP operator in 4GL reports (pages page 4-102 and page 6-50), for more information about handling blank characters in character values.)

You can insert blanks in DISPLAY or PRINT statements by using the SPACE or COLUMN operators; these are described in Chapter 4. The keyword SPACES is a synonym for SPACE.

You can use the ASCII operator in DISPLAY or PRINT statements. This takes an integer expression as its operand, and returns a single-character string, corresponding to the specified ASCII character. See Chapter 4 for details.

Non-Printable Characters

INFORMIX-4GL regards the following as the *printable* ASCII characters:

- TAB (= CONTROL-I)
- NEWLINE (= CONTROL-J)
- FORMFEED (= CONTROL-L)
- ASCII 32 (= blank) through ASCII 126 (= ~)

For the ASCII characters and their numeric codes, see Appendix G. Any other characters are *non-printable*. Character strings that include one or more non-printable characters (for example, packed fields) can be operands or returned values of character expressions. They can be stored in 4GL variables or in database columns of the CHAR, VARCHAR, and TEXT data types.

You should be aware, however, that many 4GL features for manipulating character strings were designed for printable characters only. If you create 4GL applications that use character expressions, character variables, or character columns to manipulate unprintable characters, you may encounter unexpected results. The following are examples of problems that you risk when CHAR, TEXT, and VARCHAR values include non-printable characters.

- Behavior of I/O and formatting features like the WORDWRAP attribute or the DISPLAY or PRINT statements is designed and documented for printable characters only. It may be difficult to describe or to predict the effects of data with non-printable characters with these I/O features, but the users of your application are unlikely to enjoy the results.
- Strings with unprintable characters can have unpredictable results when output to I/O devices. For example, some sequences of non-printable characters can cause terminals to misposition the cursor, clear the display, modify terminal attributes, or otherwise make the screen unreadable.

- For another example, CONTROL-D (= ASCII 4) and CONTROL-Z (= ASCII 26) in output from a report can be interpreted as logical end-of-file, causing the report to stop printing prematurely.
- If you store a zero byte (ASCII 0) in a CHAR or VARCHAR variable or column, it might be treated as a string terminator by some operators, but as data by others; and this behavior might vary between the **Rapid Development Version** and the **C Compiler Version** of INFORMIX-4GL, or even between database engines.

If you encounter these or related difficulties in processing non-printable characters, you might consider storing such values as BYTE data types.

NLS

The DBAPICODE environment variable lets computer peripherals that use a character set that is different from that of the database communicate with the database.

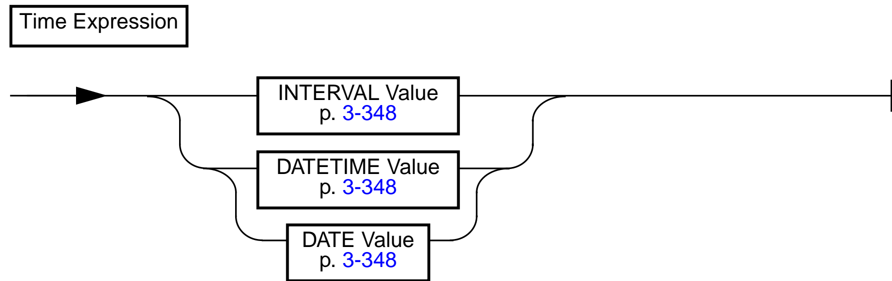
DBAPICODE specifies the character-mapping file between the peripheral and the database's character set. In NLS databases, the database character set is defined in the LC_CTYPE environment variable stored in the database locale. In non-NLS databases, the database character set is the default 8-bit character set.

Note that the NLS variable LC_COLLATE specifies the sort order for data containing characters that are outside the ASCII range but within the legal character set for the NLS locale. Note also that the NLS variable LC_CTYPE specifies a legal set of characters that can appear in identifiers.

For more information, see Appendix E, "Native Language Support Within INFORMIX-4GL."

Time Expressions

A *time expression* is a specification that INFORMIX-4GL can evaluate as a DATE, DATETIME, or INTERVAL value.

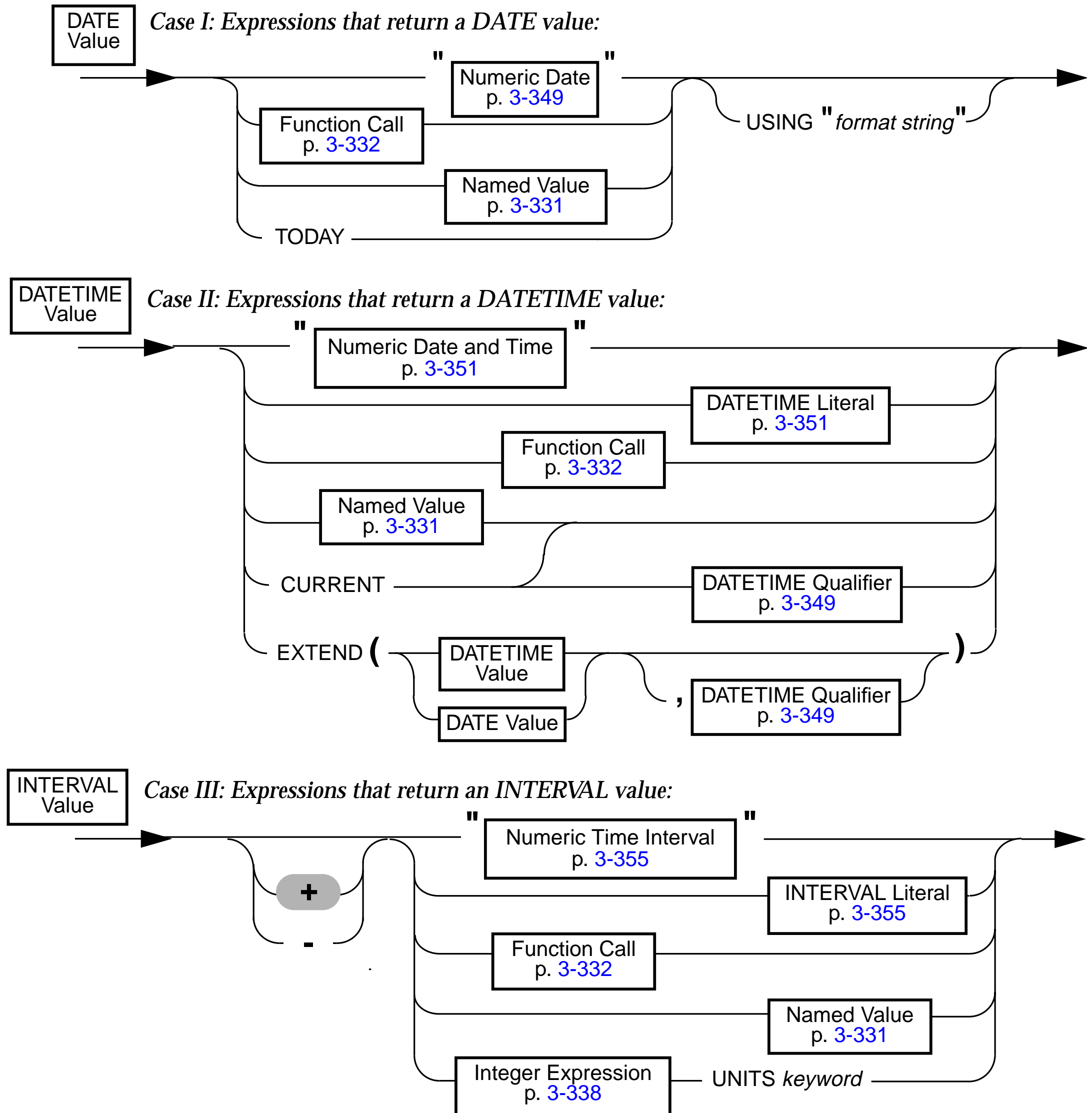


As the diagram suggests, the DATE data type is a logical subset of DATETIME. 4GL rules for arithmetic, however, are not identical for DATE and DATETIME operands (page 3-356), and formatting features like USING (page 4-91) or the FORMAT (page 5-42) and PICTURE attributes (page 5-48) treat DATETIME and DATE values differently.

The time data types are logically related because they express values as units of time. But unlike the number data types (page 3-295) or the character data types (page 3-296), within which 4GL generally supports automatic data type conversion (aside from restrictions based on truncation, overflow, or underflow), conversion among time data types is more limited. In contexts where a time expression is required, DATETIME or DATE values can sometimes be substituted for one another. INTERVAL values, however, which represent one-dimensional spans of time, cannot be converted to DATETIME or DATE values, because these represent zero-dimensional points in time.

In addition, if the declared precision of an INTERVAL value includes *years* or *months*, then automatic conversion to an INTERVAL having smaller time units (like *days*, *hours*, *minutes*, or *seconds*) is not available. See also page 3-324.

Each of the three types of time expressions has its own syntax:



format string is a quoted character string to specify a DATE display format.

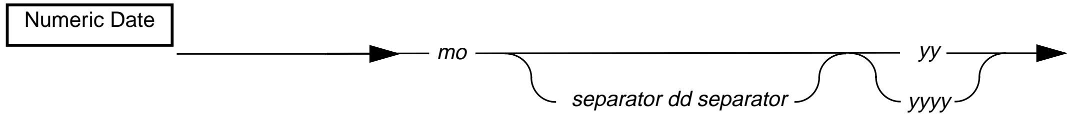
keyword is YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION.

In each case, the *function call* or *named value* must return a single value of the corresponding data type. Chapter 4 describes built-in operators like UNITS.

Numeric Date

A *numeric date* represents a DATE value as a quoted string of digits:

Numeric Date



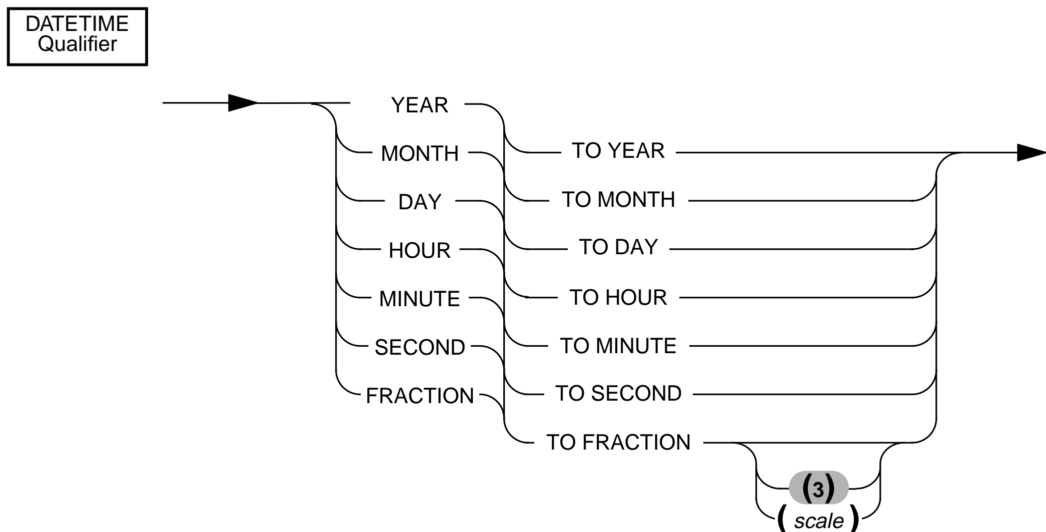
The digits must represent a valid calendar date. You can use 6 (*moddyy*) or 8 (*moddyyyy*) digits, with blank, slash (/), hyphen (**-**), or no symbol as the *separator*. Here *mo*, *dd*, and *yyyy* have the same meanings as on page 3-351. The DBDATE environment variable can change the order of time units, and can specify other separators. Like the USING operator or the FORMAT field attribute, DBDATE can also specify how 4GL displays DATE values.

If you omit the quotation marks where a DATE is expected, 4GL attempts to evaluate your specification as a literal integer or as an integer expression that specifies count of days; the result may not be useful.

DATETIME Qualifier

The *DATETIME qualifier* specifies the precision and scale of a DATETIME value. It has the same syntax as for DATETIME database columns:

DATETIME Qualifier



scale is a whole number from 1 to 5, enclosed between parentheses. This specifies the number of decimal digits for *fractions of a second*. If you omit the *scale* specification, the default scale is 3 digits.

Specify the *largest* time unit in the DATETIME as the first keyword. After the TO, specify the *smallest* time unit as the second keyword. These time units imply that these time units can be recorded in the DATETIME value:

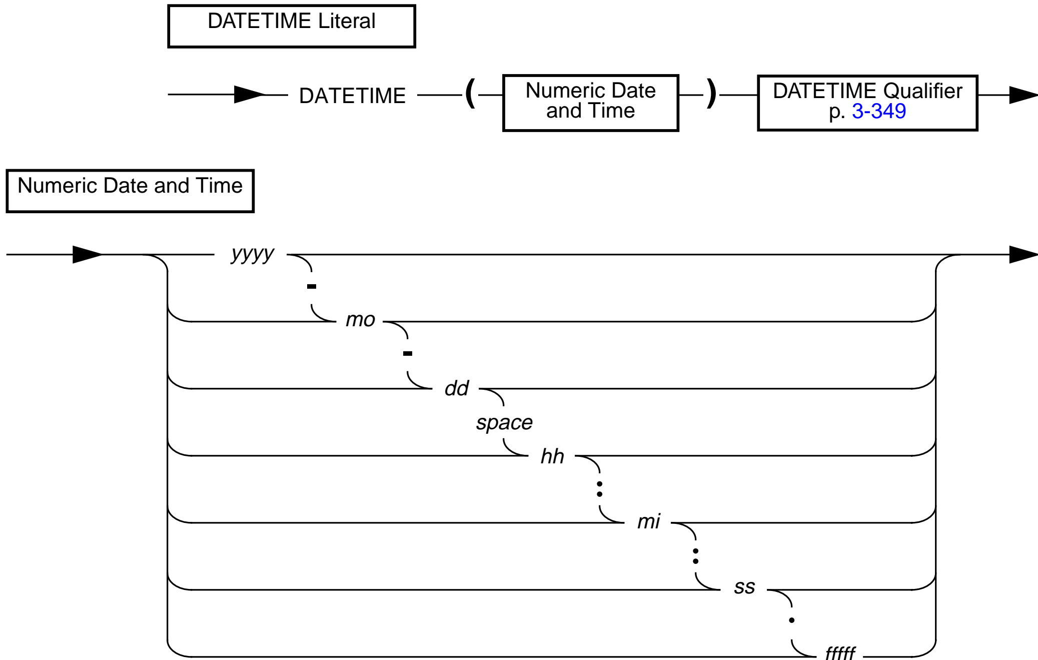
YEAR	is a year, numbered from 1 to 9999.
MONTH	is a month, numbered from 1 to 12.
DAY	is a day, numbered from 1 to 31, as appropriate to its month.
HOUR	is an hour, numbered from 0 (midnight) to 23.
MINUTE	is a minute, numbered from 0 to 59.
SECOND	is a second, numbered from 0 to 59.
FRACTION	is a fraction of a second, with up to five (5) decimal places. The default precision is three digits (thousandth of a second).

Unlike INTERVAL qualifiers, DATETIME qualifiers *cannot* specify non-default precision (except for FRACTION when it is the smallest unit in the qualifier). Here are some examples of DATETIME qualifiers:

```
YEAR TO MINUTE               MONTH TO MONTH
DAY TO FRACTION(4)           MONTH TO DAY
```

DATETIME Literal

A DATETIME *literal* is the representation of a DATETIME value as the numeric date and time, or a portion thereof, followed by a DATETIME qualifier:



dd is the number of the *day* of the month, from 1 to 31.

fffff is the *fraction of a second*, in up to 5 digits, as set by the precision specified for the FRACTION time units in the DATETIME qualifier.

hh is the *hour* (from a 24-hour clock), from 0 (= midnight) to 23.

mi is the *minute* of the hour, from 0 to 59.

mo is a number from 1 to 12, representing a *month*.

space is a *blank space* (ASCII 32), entered by pressing the SPACEBAR.

ss is the *second* of the minute, from 0 to 59.

yyyy is a number from 1 to 9999, representing a *year*. If you use only two digits, `19` is assumed as the first part of the year, as in `1993`.

An error results if you omit any required separator or includes values for units outside the range specified by the qualifier. Here are some examples:

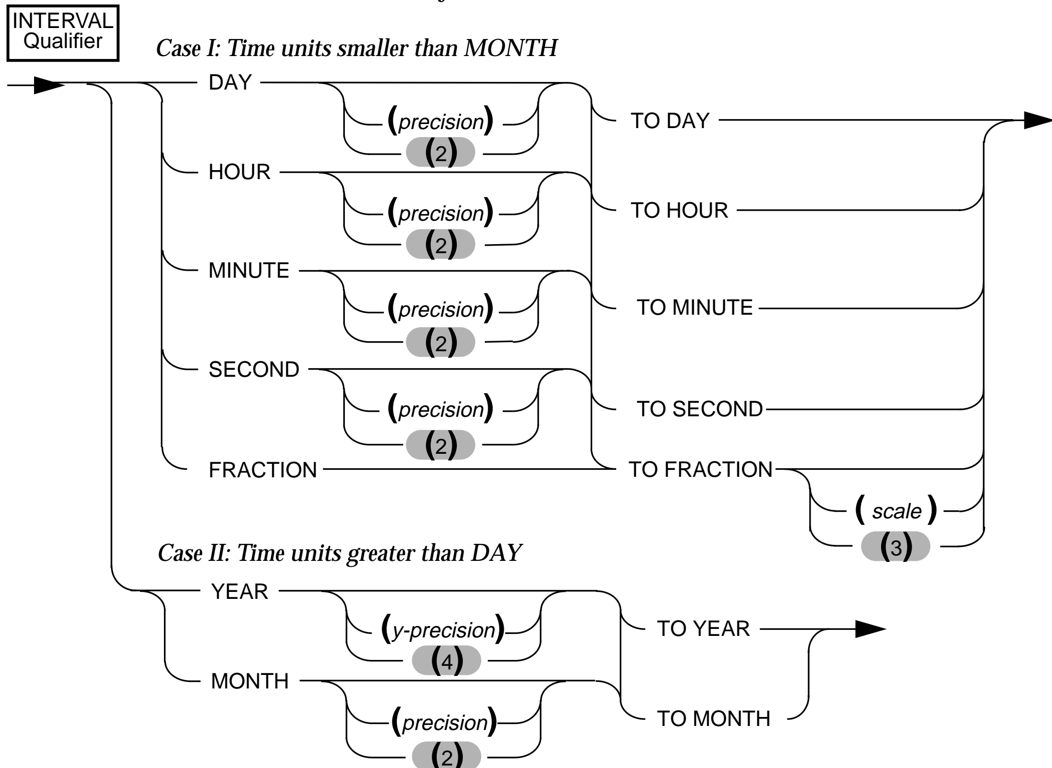
```
DATETIME (93-3-6) YEAR TO DAY
DATETIME (09:55:30.825) HOUR TO FRACTION
DATETIME (93-5) YEAR TO MONTH
```

Here is an example of a DATETIME literal used in an arithmetic expression as an operand of the EXTEND operator:

```
EXTEND (DATETIME (1993-8-1) YEAR TO DAY, YEAR TO MINUTE)
    - INTERVAL (720) MINUTE (3) TO MINUTE
```

INTERVAL Qualifier

The *INTERVAL qualifier* specifies the precision and scale of an INTERVAL value. It has the same syntax in 4GL as for INTERVAL database columns:



scale is the number of decimal digits to record fractions of a second in a span of time. The default is three digits; the maximum is five.

precision is the number of digits in the largest number of months, days, hours, minutes, or seconds that the interval can include. The default number of digits is two; the maximum is nine.

y- precision is the number of digits in the largest number of years that the interval can include. The default is four; the maximum is nine.

Specify the *largest* time unit in the INTERVAL as the first keyword. After the TO, specify the *smallest* time unit as the second keyword. If the first time unit keyword is YEAR or MONTH, the second cannot be smaller than MONTH.

The following examples of an INTERVAL qualifier are both YEAR to MONTH. The first example can record a span of up to 999 years, because 3 is the precision of the YEAR units. The second example uses the default precision for the YEAR units; it can record a span of up to 9,999 years and 11 months.

```
YEAR (3) TO MONTH
YEAR TO MONTH
```

When you intend for a value to contain only one kind of time unit, the first and last keywords in the qualifier are the same. For example, an interval of whole years that is qualified as YEAR TO YEAR can record a span of up to 9,999 years, the default precision. Similarly, the qualifier YEAR (5) TO YEAR can record a span of up to 99,999 years.

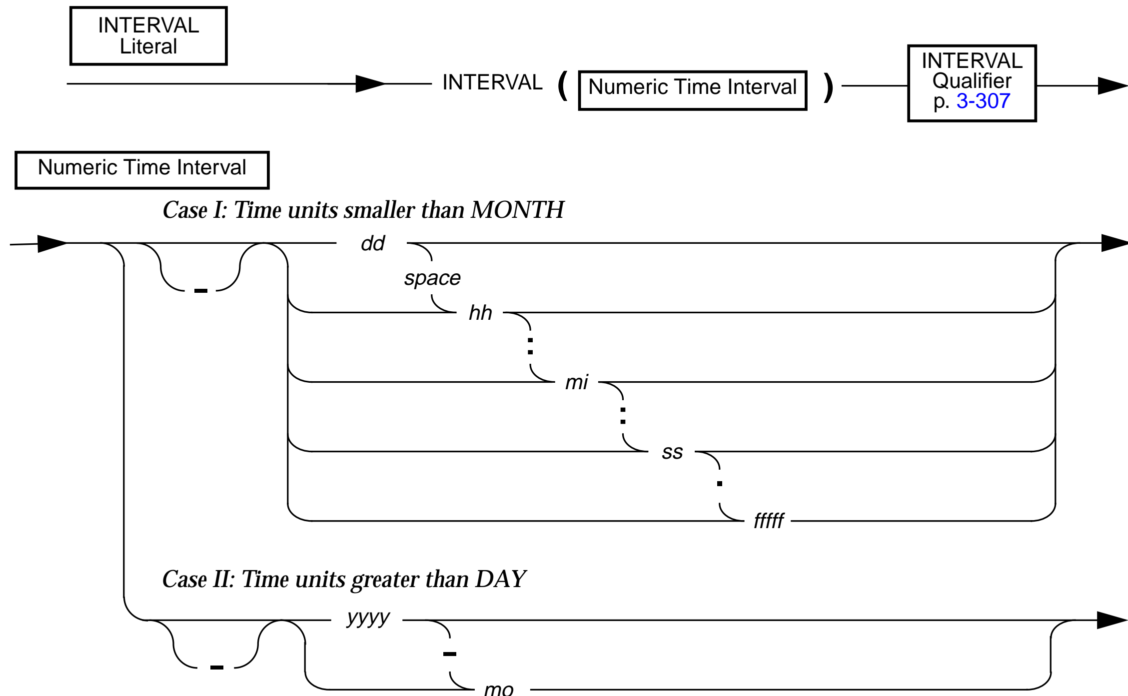
The following examples show several forms of INTERVAL qualifiers:

```
YEAR(5) TO MONTH
DAY (5) TO FRACTION(2)
DAY TO DAY
FRACTION TO FRACTION (4)
```

The option to specify a non-default *precision* or *y-precision* (as distinct from the *scale*) is a feature that INTERVAL variables do not share with DATETIME variables. An error results if you attempt to do this when you declare a DATETIME variable, specify DATETIME literal, or call the EXTEND operator.

INTERVAL Literal

An *INTERVAL literal* represents a span of time as a numeric representation of its chronological units, followed by an INTERVAL qualifier.



dd	is the number of days.
fffff	is the fraction of a second in up to five digits, depending on the precision of the fractional portion in the INTERVAL qualifier.
hh	is the number of hours.
mi	is the number of minutes.
mo	is the number of months, in two digits.
space	is a *blank space* (= ASCII 32), made by pressing the spacebar.
ss	is the number of seconds.
yyyy	is the number of years.

For all time units except *years* and *fractions*-of-a-second, the maximum number of digits allowed is two, unless this is the first time unit, and the precision is specified differently by the INTERVAL qualifier. (For *years*, the default maximum number of digits is four, unless this is the first time unit, and some other precision is specified by the INTERVAL qualifier.

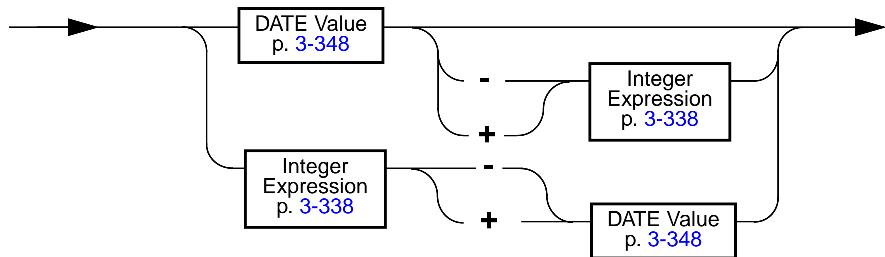
Neither the numeric values nor the qualifier can combine units of time that are smaller than *month* with *month* or *year* time units.

An error results if an INTERVAL literal omits any required field separator, or includes values for units outside the range specified by the field qualifiers. Some examples of INTERVAL literal values follow:

```
INTERVAL (3-6) YEAR TO MONTH
INTERVAL (09:55:30.825) HOUR TO FRACTION
INTERVAL (40-5) DAY TO HOUR
```

Arithmetic Operations on Time Values

Time expressions can be operands of some arithmetic operators. These expressions return a DATE value:

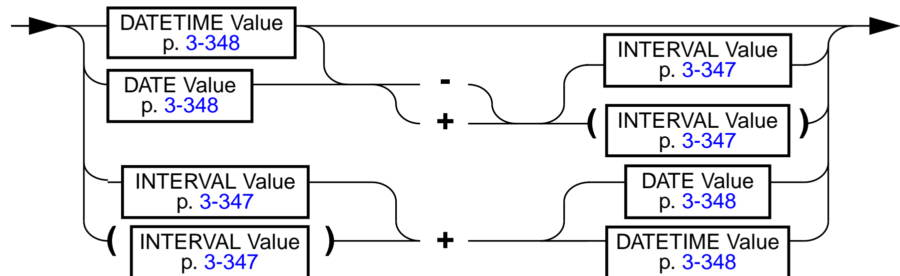


All the other binary arithmetic operators (page 3-341) also accept DATE operands, equivalent to the count of days since December 31, 1899, but the values returned (except from a DATE expression as the left-hand MOD operand) are meaningless in most applications.

Note: *DATE and DATETIME values have no true zero point; they lie on "interval" scales. Such scales support addition and subtraction, as well as relational operators (page 3-334), but multiplication, division, and exponentiation are undefined.*

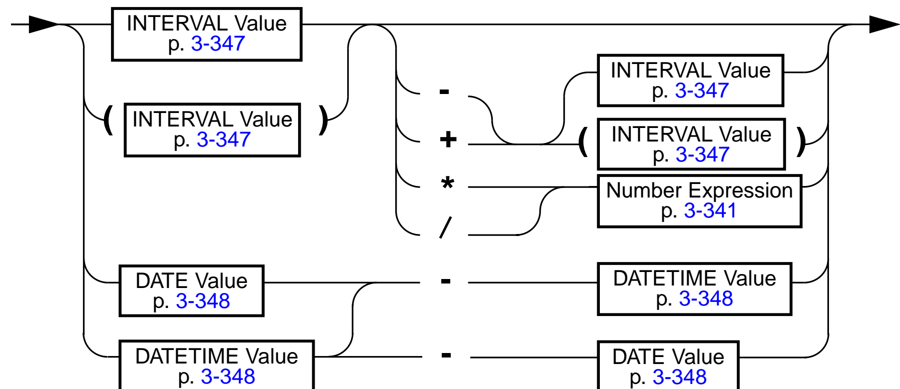
The difference between two DATE values is an INTEGER value, representing the positive or negative number of *days* between the two calendar dates. You must explicitly apply the UNITS DAY operator to the difference between DATE values, if you wish to store the result as an INTERVAL value.

This is the syntax for arithmetic expressions that return a DATETIME value:



Do not write expressions that specify the sum (+) of two DATE or DATETIME values, nor a difference (–) whose second operand is a DATE or DATETIME value, and whose first operand is an INTERVAL value.

This is the syntax for arithmetic expressions that return an INTERVAL value:



The difference between two DATETIME values (or a DATETIME and a DATE value, but *not* two DATE values) is an INTERVAL value. If the operands have different qualifiers, the result has the qualifier of the first operand.

An expression cannot combine an INTERVAL value of precision in the range YEAR to MONTH with another of precision in the DAY to FRACTION range. Similarly, you cannot combine an INTERVAL value with a DATETIME or DATE value that has different qualifiers. You must use the EXTEND function to change the DATE or DATETIME qualifier to match that of the INTERVAL.

If the first operand of an arithmetic expression is a time expression that includes the UNITS operator, you must enclose that operand in parentheses.

If any component of an time expression is a NULL value, INFORMIX-4GL evaluates the entire expression as a NULL value, unless the NULL value is an operand of the IS NULL or IS NOT NULL keywords.

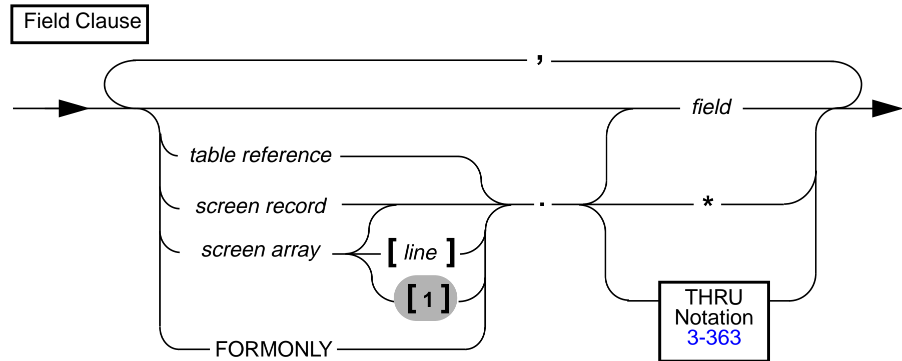
Relational Operators and Time Values

Time expression operands of *relational operators* (page 3-334) follow these rules:

- Comparison **x** < **y** is TRUE when **x** is a *briefer* INTERVAL span than **y**, or when **x** is an *earlier* DATE or DATETIME value than **y**.
- Comparison **x** > **y** is TRUE when **x** is a *longer* INTERVAL span than **y**, or when **x** is a *later* DATE or DATETIME value than **y**.
- You cannot mix INTERVAL operands with DATE or DATETIME operands, but you can compare DATE and DATETIME expressions with each other.

Field Clause

The field clause specifies one or more screen fields or screen records.



field	is a field name, as declared in the ATTRIBUTES section of the form specification file.
line	is an integer expression, enclosed within brackets, to specify a record within the *screen array.* Here 1 ≤ *line* ≤ *size*, for *size* the array size that is declared in the INSTRUCTIONS section. If you omit the [*line*] specification, the default is the *first* record.
screen array	is the 4GL identifier that you declared for a screen array in the INSTRUCTIONS section of the form specification file.
screen record	is the 4GL identifier that you declared for a screen record, or else a *table reference* (as the name of a default screen record).
table reference	is the name, alias, or synonym of a database table or view.

Usage

A *table reference* cannot include table qualifiers. You must declare an alias in the form specification file, as described on page 5-19, for any table reference that requires a qualifying prefix (such as *database*, *server*, or *owner*). Here the FORMONLY keyword (page 5-24) acts like a table reference for fields that are not associated with a database column.

You can use the asterisk (*) symbol to specify every field in a screen record.

Some contexts, such as the NEXT FIELD clause, support only a single-field subset of this syntax. In these contexts, the THRU or THROUGH keyword, asterisk notation, and comma-separated list of field names are not valid.

You can specify one or more of the following in the field clause:

- A field name without qualifiers (*field*) if this name is unique in the form
- A field name, qualified by a table reference (FORMONLY **.** *field* or *table.field*)
- An individual member of a screen record (*record* **.** *field*)
- An individual field within a screen array (*array* **[** *line* **] .** *field*)
- A set of consecutive fields in a screen record (by the THRU notation)
- An entire screen record (*record* **.** *)
- The first screen record in a screen array (*array* **.** *)
- Any entire record within a screen array (*array* [*line*] **.** *)

The FIELD_TOUCHED() operator in a CONSTRUCT or INPUT statement recognizes a subset of this *field clause* syntax; it does not support the **[** *line* **]** notation to specify an individual screen record within a screen array.

The *field list* of a SCREEN RECORD specification in the INSTRUCTIONS section of a screen form can include the THRU or THROUGH keywords (page 3-363). Chapter 4 describes how to declare screen records and screen arrays.

The following INPUT statement illustrates how to specify a field name:

```
INPUT p_customer.fname, p_customer.lname FROM fname, lname
```

The following SCROLL statement moves the displayed values in all the fields the **s_orders** screen array downwards by two lines. Any values are cleared from the first two screen records; any values in the two screen records that are closest to the bottom of the 4GL screen or other 4GL windows are no longer visible:

```
SCROLL s_orders.* DOWN 2
```

The next SCROLL statement moves the displayed values in two of the fields of the **s_orders** screen array towards the top of the 4GL screen for every screen record. Any other fields of the **s_orders** array are not affected:

```
SCROLL s_orders.stock_num, s_orders.unit_descr UP 2
```

The following CLEAR statement clears one record of a screen array. In this example, the integer value of the **idx** variable determines which screen record is cleared:

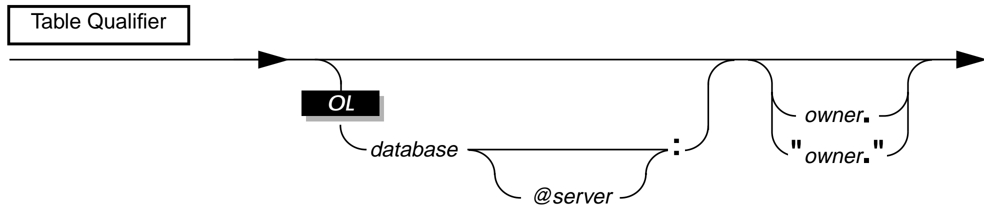
```
CLEAR s_items[idx].*
```

References

CLEAR, CONSTRUCT, DISPLAY, INPUT, INPUT ARRAY, SCROLL, THRU

Table Qualifiers

Statements that reference database tables, views, or synonyms (either alone, or as qualifiers of database column names) can include *table qualifiers*:



database is the name of a database containing the table, view, or synonym.

owner is the login name of the owner of the table, view, or synonym whose identifier immediately follows the table qualifier.

server is the name of the host system where *database* resides. Blank spaces are not valid after the @ symbol.

Usage

Table qualifiers can appear in SQL and other **4GL** statements, and in table alias declarations (page 5-19) in the TABLES section of form specifications. You cannot, however, prefix a table alias or a field name with a table qualifier. Except in table alias declarations within the TABLES section, you cannot include table qualifiers anywhere in a form specification file.

Owner Naming

The qualifier can specify the login name of the *owner* of the table. You must specify *owner* if *table.column* is not a unique identifier within its database.

ANSI

In an ANSI-compliant database, you must qualify each table name with that of the *owner* of the table (*owner.table*). The only exception is that you can omit the *owner* prefix for any tables that you own. For example, if Les owns table **t1**, you own table **t2**, and Sasha owns table **t3**, then you could use the following statement to reference three columns in those tables:

```
VALIDATE var1, var2, var3 LIKE les.t1.c1, t2.c2, sasha.t3.c3
```

You can include the owner name in a database that is not ANSI-compliant. If the *owner* is incorrect, however, 4GL generates an error. For more information, see the section “Owner Naming” in the *Informix Guide to SQL: Reference*.

Database References

The LIKE clause of **4GL** statements like DEFINE, INITIALIZE, and VALIDATE can use this *database*: or *database*@*server*: notation in table qualifiers to specify tables in a database other than the default database (page 3-59). Without such qualifiers, **4GL** looks for the table in the default database. Even if the table qualifier includes a database reference, however, the LIKE clause will fail unless you also include a DATABASE statement before the first program block in the same module to specify a default database.

The *current database* is the database specified by the most recently executed DATABASE statement in a MAIN or FUNCTION program block in the same module. 4GL programs can include SELECT statements that query a table in an INFORMIX-OnLine database that is not the current database, but they cannot insert, update, nor delete rows from any table that is not in the current database.

If the current database is supported by the INFORMIX-OnLine database engine, a table reference can also include @*server* to specify the name of another host system on which a table resides.

```
LOAD FROM "fyl" INSERT INTO dbas@hostile:woody.table42
```

SE

Only the databases stored in your current directory, or in a directory specified in your DBPATH environment variable, are recognized. Table qualifiers cannot include references to an INFORMIX-SE database.

ANSI

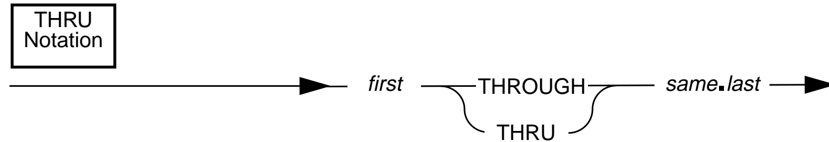
If the current database is ANSI-compliant, a run-time error results if you attempt to query a remote database that is not ANSI-compliant.

References

DATABASE, DEFINE, INITIALIZE, LOAD, VALIDATE, UNLOAD

THRU or THROUGH Keywords and .* Notation

To list consecutive set members in the same order as in their declaration, you can use the `.*` notation to specify the entire set, or you can use the keyword THRU (or THROUGH, its synonym) to specify a subset of consecutive items:



first	is the name of some member variable or field of the record.
last	is a variable or field that was declared later than *first*.
same	is the name of the same record or table that qualified *first*.

Usage

These notational devices in 4GL statements can simplify lists of structured sets of *fields* of a screen record, or *member variables* of a program record. (The *columns* of a database table can be referenced by the asterisk notation, but you cannot use THRU nor THROUGH to specify a partial list of columns.)

If the ALTER TABLE statement has changed the order, the names, the data types, or the number of the columns in *table* since you compiled your program, then you might need to modify your program and its screen forms that reference that table before you can use these notational devices.

The notation *record.member* refers to an individual member variable of a 4GL program record, or a field of a 4GL screen record. The *record*`.*` notation refers to the entire program record or screen record. Here *record* can be the name, alias, or synonym of a table or view, or the name of a program record or of a screen record, or the FORMONLY keyword.

The THRU (or equivalently, THROUGH) notation can specify a partial list of the members of a program record or screen record. The notation *record.first* THRU *record.last* refers to a consecutive subset of members of the record, from *first* through *last* inclusive, where *first* appears sooner than *last* in the data type declaration of a program record, or else in the ATTRIBUTES section of the form specification file (for screen records).

These notations are a shorthand for writing out a full or partial list of set members, with comma (,) separating individual items in the list; this is the form to which 4GL expands these notations. Here are two examples:

```
INITIALIZE pr_rec.member4 THRU pr_rec.member8 TO NULL
DISPLAY pr_rec.* TO sc_rec.*
```

The INITIALIZE statement above sets to NULL the values of 4GL variables **pr_rec.member4**, **pr_rec.member5**, **pr_rec.member6**, **pr_rec.member7**, and **pr_rec.member8**. The DISPLAY statement lists the entire record **pr_rec** in the screen fields that comprise the screen record **sc_rec**.

The order of record members within the expanded list is the same order that they had when they were declared, from *first* to *last*. For a *screen record*, this is the order of their field descriptions in the ATTRIBUTES section. For example, suppose that the following appeared in the form specification file:

```
ATTRIBUTES
...
f002=tab3.aa;
f003=tab3.bb;
f004=tab3.cc;
f005=tab2.aa;
f006=tab2.bb;
f007=tab1.aa;
f008=tab1.bb;
f009=tab1.cc;
...
INSTRUCTIONS
SCREEN RECORD sc_rec (tab3.cc THRU tab1.bb)
```

The previous fragment of a form specification file implies the following ordered list of field names within the screen record **sc_rec**:

```
tab3.cc     tab2.aa     tab2.bb     tab1.aa     tab1.bb
```

Note: *The order of fields in the screen record depends on the physical order of field descriptions in the ATTRIBUTES section, and on the SCREEN RECORD specification. The Form Compiler ignores the physical arrangement of fields in the screen layout, the order of table names in the TABLES section, the CONSTRAINED and UNCONSTRAINED keywords of the OPTIONS statement, and the lexicographic order of the table names or field names when it processes the declaration of a screen record. For more information about default and non-default screen records, see* *page 5-63.*

Restrictions on the THRU, THROUGH, and .* Notations

The THRU, THROUGH, or .* notation can appear in any list of columns, fields, or member variables, with the following exceptions:

- You cannot use THRU nor THROUGH in reference to *columns* of database tables. There is no shorthand for a partial listing of columns of a table.
- You cannot use THRU nor THROUGH to indicate a partial list of *screen record* members while the program displays or enters data in a form.
- You cannot use THRU, THROUGH, nor .* in a quoted string to specify the *variable list* of a SELECT or INSERT clause in the PREPARE statement.
- You cannot use THRU, THROUGH, or the .* notation to reference a program record that contains an *array* member. (But these notations can specify all or part of a record that contains records as members.)
- An exception to the general rule of .* expanding to a list of all column names occurs when .* appears in an UPDATE statement. Here any columns of the SERIAL data type are excluded from the expanded list. For example, the following UPDATE statement:

  ```
  UPDATE table1 SET table1.* = program_rec.*
  ```

  is equivalent to the expanded syntax:

  ```
  UPDATE table1 SET table1.col1 = program_rec.member1,
  table1.col2 = program_rec.member2, ...
  ```

  and so forth to the last column, but with any SERIAL column omitted.

References

CLEAR, CONSTRUCT, DISPLAY, FUNCTION, INPUT, INPUT ARRAY, REPORT, SCROLL

Chapter

4

Built-In Functions and Operators

Functions in 4GL Programs 5
- Built-In 4GL Functions 5
- Built-In SQL Functions 6
- C Functions 6
- ESQL/C Functions 7
- Programmer-Defined 4GL Functions 7
- Invoking Functions 8

Operators of 4GL 10

Syntax of Built-In Functions and Operators 11
- Aggregate Report Functions 13
  - The GROUP Keyword 14
  - The WHERE Clause 14
  - The MIN( ) and MAX( ) Functions 14
  - The AVG( ) and SUM( ) Functions 14
  - The COUNT (*) and PERCENT (*) Functions 14
- ARG_VAL( ) 16
- Arithmetic Operators 18
  - Unary Arithmetic Operators 19
  - Binary Arithmetic Operators 19
  - Exponentiation (**) Operator 21
  - Modulus (MOD) Operator 22
  - Multiplication (*) and Division (/) Operators 22
  - Addition (-) and Subtraction (+) Operators 22
- ARR_COUNT( ) 24
- ARR_CURR( ) 26
- ASCII 28

Boolean Operators 30
Logical Operators 31
Boolean Comparisons 31
Relational Operators 32
The NULL Test 33
The LIKE and MATCHES Operators 33
Set Membership and Range Tests 35
CLIPPED 38
COLUMN 40
CURRENT 42
DATE 44
DATE( ) 45
DAY( ) 46
DOWNSHIFT( ) 47
ERR_GET( ) 48
ERR_PRINT( ) 49
ERR_QUIT( ) 50
ERRORLOG( ) 51
EXTEND( ) 53
FGL_DRAWBOX( ) 56
FGL_GETENV( ) 58
FGL_KEYVAL( ) 60
FGL_LASTKEY ( ) 62
FIELD_TOUCHED( ) 64
GET_FLDBUF( ) 66
INFIELD( ) 69
LENGTH( ) 71
LINENO 73
MDY( ) 74
MONTH( ) 75
NUM_ARGS( ) 76
PAGENO 77
SCR_LINE( ) 78
SET_COUNT( ) 80
SHOWHELP( ) 81
SPACE 82
SQLEXIT( ) 83
STARTLOG( ) 84
TIME 86
TODAY 87
UNITS 89
UPSHIFT( ) 90

USING 91
USING Operator Examples 96
WEEKDAY( ) 100
WORDWRAP 102
YEAR( ) 104

Functions in 4GL Programs

In **4GL**, a *function* is a named collection of statements that performs a task. (In some programming languages, terms like *method*, *subroutine*, or *procedure* correspond to a "function" in **4GL**.) If you need to repeat the same series of operations, you can call the same function several times, rather than specify the same steps for each repetition. This construct supports the structured programming design goal of segmenting source code modules into logical units, each of which has only a single entry point and controlled exit points.

The FUNCTION statement (page 3-111) can define functions. **4GL** programs can invoke the following types of functions:

- Programmer-defined **4GL** functions
- **4GL** built-in functions
- **SQL** built-in functions
- **C** functions
- **ESQL/C** functions (*if you have the* **INFORMIX-ESQL/C** *product*)

Programmer-defined **4GL** functions are described briefly on page 4-7, and in greater detail on page 3-111. The other types of functions that you can call from a **4GL** program are briefly discussed on the next two pages.

Built-In 4GL Functions

The *built-in functions* of 4GL are predefined functions that support features of the INFORMIX-4GL language. Except for the fact that no FUNCTION definition is required, built-in functions behave exactly like the **4GL** functions that you define with the FUNCTION statement:

- You can invoke them with the CALL statement. (If they return a single value, they can appear without CALL in **4GL** expressions.)
- They require parentheses (**( )**), even if the argument list is empty.
- You cannot invoke them from SQL statements.
- You can invoke them from a C program.

If you use the FUNCTION statement to define a function with the same name as a built-in function, then your program cannot invoke the built-in function. Each of the following **4GL** built-in functions is described in this chapter.

ARG_VAL(*int-expr*)
ARR_COUNT( )
ARR_CURR(*char-expr*)
DOWNSHIFT(*char-expr*)
ERR_GET(*int-expr*)
ERR_PRINT(*int-expr*)
ERR_QUIT(*int-expr*)
ERRORLOG(*char-expr*)
FGL_DRAWBOX(*nlines, ncols, begy, begx, color*)
FGL_GETENV(*char-expr*)
FGL_KEYVAL(*char-expr*)
FGL_LASTKEY( )
LENGTH(*char-expr*)
NUM_ARGS( )
SCR_LINE( )
SET_COUNT(*int-expr*)
SHOWHELP(*int-expr*)
SQLEXIT( )
STARTLOG("*filename*.*filetype*")
UPSHIFT(*char-expr*)

Each argument of FGL_DRAWBOX( ) is an integer expression (page 3-338), except *color*, which can also be a keyword to specify a color (page 3-290). You can also use *4GL aggregate functions* (page 4-13), but only in REPORT program blocks. These aggregates, which include AVG( ), COUNT(*), MAX( ), MIN( ), PERCENT(*), and SUM( ), cannot be operands in **4GL** expressions.

Built-In SQL Functions

Informix database engines support built-in SQL functions, some of which have the same names as built-in **4GL** functions or operators. The built-in SQL functions can appear only in **SQL** statements, but not in other 4GL statements. (The *Informix Guide to SQL: Reference* describes SQL functions.)

Note: *Versions of Informix database engines later than 4.1 support stored procedures. These resemble functions that are executed by the database engine. This version of 4GL does not provide direct support for the EXECUTE PROCEDURE statement of SQL. You must use the PREPARE statement if you want to execute a stored procedure from a program that you create with this release of 4GL.*

C Functions

You can use the CALL statement or an expression to invoke properly-written C language functions within an **INFORMIX**-**4GL** program. Such functions are often helpful for specialized tasks that are not easily written in **4GL**, such as processing binary I/O. For information on the application program interface (API) of **4GL** to the C programming language, see Appendix C.

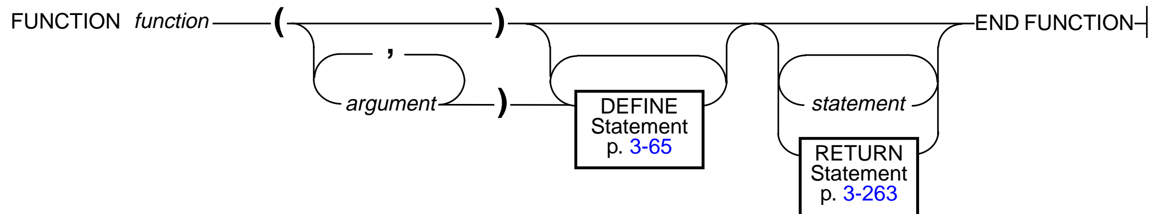
Unlike **INFORMIX**-**4GL** identifiers, names of C functions are case-sensitive. They must typically appear in *lowercase* letters within the function call.

ESQL/C Functions

If you have the **INFORMIX-ESQL/C** product, your **4GL** program can also call compiled **ESQL/C** functions that you write, as well as **ESQL/C** library functions. See "Running Programs that Call C Functions" on page 1-70.

Programmer-Defined 4GL Functions

The FUNCTION program block begins with the FUNCTION keyword and ends with the END FUNCTION keywords. These enclose a program block of the **4GL** statements that compose the function, and that are executed when the function is invoked. This is the syntax of the FUNCTION statement:



argument is the name of a formal argument to the function.

function is the name that you declare for the function.

statement is an SQL statement or other **4GL** statement.

The left-hand portion of this diagram, including the identifier of the function and the list of formal arguments, is sometimes called the *function prototype* (page 3-112). This resembles the prototype of a report (page 6-6).

Names of functions must be unique among the names of functions, reports, and global variables within the program, and cannot be the same as any of the formal arguments of the same function. See "FUNCTION" on page 3-111 for details of how to define **4GL** functions.

The right-hand portion of this diagram, including the declarations of formal arguments and of local variables, and the *statement* block, is sometimes called the FUNCTION *program block*. This can include any executable statement of SQL or 4GL except the report execution statements (page 3-14). The entire FUNCTION program block must be defined within a single source module.

No other FUNCTION, REPORT, nor MAIN program block can be included in a FUNCTION definition, but it can include statements that produce a report, or call a function, or execute a RUN statement that invokes another program.

Invoking Functions

Except for SQL functions, which are called in SQL expressions (page 3-330), 4GL programs can use the CALL statement (page 3-16) to invoke functions. In some contexts, however, you can also call functions implicitly:

- If a function returns a single value, then you can invoke the function simply by specifying its name (and any required arguments) within an expression where a value of the returned data type is valid.
- The exception-handling features of **4GL** can automatically invoke a function that you specify in the CALL clause of the WHENEVER statement.

Passing Arguments and Returning Values

The program block containing the CALL statement or expression that invokes a function is called the *calling routine*. Functions can receive information from (and return values to) the calling routine. In the typical case where this is a different program block, values from the calling routine and from other program blocks are visible to the function only through global or module variables (page 3-65), or through the argument list of the calling statement.

For most data types, the RETURN statement (page 3-263) in the function and RETURNING clause (page 3-19) of the CALL statement specify any values that the function returns to the calling routine. This mechanism for communication between the function and its calling routine is called *passing by value.*

Arguments of blob data types BYTE or TEXT are processed in a different way, called *passing by reference*. The BYTE or TEXT variables appear in the argument list of the calling statement, but what is passed to the function is a pointer to the variables. The RETURN statement and RETURNING clause cannot include blob variables. (The built-in 4GL functions that are described in this chapter all pass their arguments by value, rather than by reference.)

Invoking SQL Functions

You can invoke predefined SQL functions and operators in **4GL** programs, but only within SQL statements. (See the description of function expressions in the *Informix Guide to SQL: Reference* for information about SQL functions.) For example, the USER operator of SQL can appear in a SELECT statement:

```
DEFINE usr_id CHAR(9)
...
SELECT USER INTO usr_id FROM systables WHERE tabid = 1
```

Note: *Some built-in* **4GL** *functions and operators have the same names as SQL functions or operators. For example,* CURRENT, DATE( ), DAY( ), EXTEND( ), LENGTH( ), MDY( ), MONTH( ), WEEKDAY( ), YEAR( ), *and the relational operators (**page 4-32**) are features of both* **4GL** *and SQL. You will generally encounter a compile-time or link-time error, however, if a statement that is not an SQL statement references an SQL function or operator that is not also a* **4GL** *function or operator.*

Built-in SQL functions and operators like USER cannot appear in other 4GL statements, however, that are not SQL statements. If a program requires the functionality of USER in a non-SQL statement like PROMPT, for example, then you must first use FUNCTION to define an equivalent **4GL** function:

```
FUNCTION get_user()
   DEFINE uid LIKE informix.sysusers.username
   SELECT USER INTO uid FROM informix.systables
      WHERE tabname = "systables"
      -- row is sure to exist and to be singular
   RETURN uid
END FUNCTION
```

To require no cursor, the SELECT statement in this example must be written so that it returns only one row. Here the **get_user( )** function selects the row of **systables** that names itself, because this row it is sure to exist and to be unique. (The owner name "**informix**" is required to reference tables of the system catalog only in an ANSI-compliant database, but it is valid in any SQL database where it is an owner name.)

Operators of 4GL

The *operators* support features of **INFORMIX-4GL** that in earlier releases were called "*built-in functions."* The operators are different in several ways from **4GL** functions:

- Except for GET_FLDBUF( ), the CALL statement cannot invoke them.
- Some operators can take special non-alphanumeric symbols as operands.
- Some can be used in SQL statements.
- You cannot reference them from a C program.

Despite these differences, the operators are described in this chapter, since they resemble the built-in **4GL** functions in their syntax and behavior. Operators that return a single value can be operands in expressions.

You can use the FUNCTION statement to define a 4GL function with the same name as an operator. In this case, only the operator, not the function, is visible as an operand in a 4GL expression. For example:

```
let dt = mdy(1,2,3)    --built-in MDY() operator
call mdy(1,2,3)        --programmer-defined MDY() function
```

You can use the CALL statement, however, to invoke a function that has the same name as an operator.

The following operators of 4GL are described in this chapter:

ASCII *int-expr*
char-expr CLIPPED
COLUMN *integer*
CURRENT
CURRENT *qualifier*
DATE
DATE (*date-expression*)
DAY(*date-expression*)
EXTEND(*value*)
EXTEND(*value*, *qualifier*)
FIELD_TOUCHED(*field-list*)
GET_FLDBUF(*field-list*)
INFIELD(*field*)
LENGTH(*char-expr*)
LINENO
MDY(*int-expr, int-expr, int-expr*)
MONTH(*date-expression*)
PAGENO
int-expr SPACE
int-expr SPACES
TIME
TODAY
int-expr UNITS *time-keyword*
expression USING *format-string*
WEEKDAY (*date-expression*)
YEAR (*date-expression*)

Note: *These operators and additional arithmetic, logical, and relational operators are included in this chapter as a convenience, so that you can find syntax articles without classifying a given feature as a function or as an operator.*

Syntax of Built-In Functions and Operators

Operators that are represented by non-alphabetic symbols are grouped in this chapter under the headings "Arithmetic Operators" on page 4-18 and "Relational Operators" on page 4-32.

The relational operators can appear in expressions whose returned values are either TRUE, FALSE, or NULL. The section "Boolean Operators" on page 4-30 describes the operators of 4GL that can return only these Boolean values. The various aggregates that can appear in a REPORT definition are themselves aggregated in the section "Aggregate Report Functions" on page 4-13.

For a general discussion of **INFORMIX-4GL** operators, see page 3-327.

Sections that follow describe these arithmetic and relational operators:

Symbol	Description	Page	Symbol	Description	Page
+	Addition	4-22	<	Less than	4-31
/	Division	4-22	<=	Not greater than	4-31
**	Exponentiation	4-21	= *or* ==	Equal to	4-31
MOD	Modulus	4-21	!= *or* <>	Not equal to	4-31
*	Multiplication	4-22	>=	Not less than	4-31
-	Subtraction	4-22	>	Greater than	4-31

Additional sections describe these built-in functions and operators of 4GL:

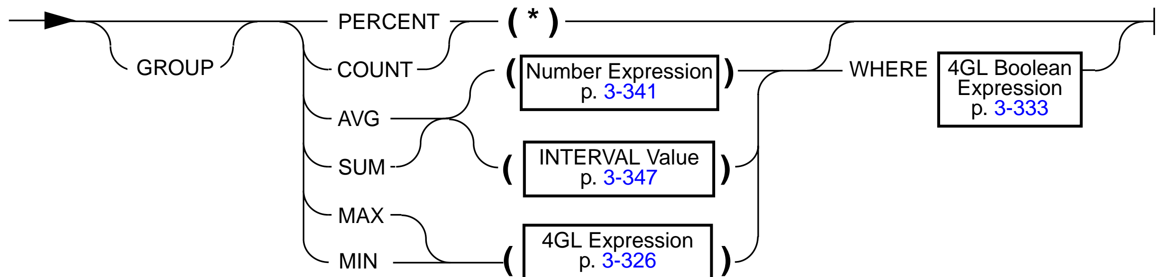
	Built-in Functions	Page		Operators	Page
	ARG_VAL( )	4-16		AND	4-30
	ARR_COUNT( )	4-24		ASCII	4-28
	ARR_CURR( )	4-26	‡	BETWEEN . . . AND	4-35
†	AVG( )	4-13		CLIPPED	4-38
†	COUNT(*)	4-13		COLUMN	4-40
	DOWNSHIFT( )	4-47		CURRENT	4-42
	ERR_GET( )	4-48		DATE	4-44
	ERR_PRINT( )	4-49		DATE( )	4-45
	ERR_QUIT( )	4-50		DAY( )	4-46
	ERRORLOG( )	4-51		EXTEND( )	4-53
	FGL_DRAWBOX( )	4-56		FIELD_TOUCHED( )	4-64
	FGL_GETENV( )	4-58		GET_FLDBUF( )	4-66
	FGL_KEYVAL( )	4-60	‡	IN( )	4-35
	FGL_LASTKEY( )	4-62		INFIELD( )	4-69
	LENGTH( )	4-71		IS NULL	4-33
†	MAX( )	4-13		LIKE	4-33
†	MIN( )	4-13	†	LINENO	4-72
	NUM_ARGS( )	4-76		MATCHES	4-33
†	PERCENT(*)	4-13		MDY( )	4-74
	SCR_LINE( )	4-78		MONTH( )	4-75
	SET_COUNT( )	4-80		NOT	4-30
	SHOWHELP( )	4-81		OR	4-30
	SQLEXIT( )	4-83	†	PAGENO	4-77
	STARTLOG( )	4-84		SPACE or SPACES	4-82
†	SUM( )	4-13		TIME	4-86
	UPSHIFT( )	4-90		TODAY	4-87
				UNITS	4-89
				USING	4-91
				WEEKDAY( )	4-100
			†	WORDWRAP	4-102
				YEAR( )	4-104

† *Valid only in the* FORMAT *section of a* REPORT *program block.*

‡ *Valid only in the* COLOR *attribute of a form specification, and in SQL statements.*

Aggregate Report Functions

Each *aggregate report function* of 4GL returns a value summarizing data from all the input records, or from a specified group of input records. The 4GL report aggregates are not valid outside of a REPORT program block.



Usage

The 4GL report aggregates resemble the SQL aggregates that can appear in SELECT or DELETE statements, but their syntax is not identical; see page 4-15. Aggregate report functions cannot appear as operands of 4GL expressions, and cannot be nested. That is, no expression within a report aggregate can include a report aggregate.

An error typically occurs if you attempt to use the name of an aggregate as an identifier. Programmer-defined functions to calculate the same statistics can be invoked from within or outside of REPORT definitions, but you must declare other names for such functions.

Variables of large or structured data types (page 3-296) cannot be arguments to these functions. You can, however, specify the name of a simple variable (page 3-294) that is a member of a record, or that is an element of an array.

AVG( ), SUM( ), MIN( ), and MAX( ) ignore records with NULL values for their argument, but each returns NULL if all records have a NULL value.

If an aggregate value that depends on all records of the report appears anywhere except in the ON LAST ROW control block, then each variable in that aggregate or WHERE clause must also appear in the list of formal arguments of the report. (Examples of aggregates that depend on all records include using GROUP COUNT(*) anywhere in a report, or using any aggregate without the GROUP keyword anywhere outside the ON LAST ROW control block.)

4GL stores intermediate results for aggregates in temporary tables. An error results if no database is open, because the temporary table cannot be created.

The GROUP Keyword

This optional keyword causes the aggregate function to include data only for a *group* of records that have the same value on a variable that you specify in an AFTER GROUP OF control block.

An aggregate can include the GROUP keyword only within an AFTER GROUP OF control block. If you need the value of a GROUP report aggregate elsewhere, you must use the LET statement within the AFTER GROUP OF control block to store the value in a variable of appropriate scope of reference.

The WHERE Clause

The optional WHERE keyword selects among the records passed to the report, including only those for which a Boolean expression (page 3-333) is TRUE. Conditional aggregates are calculated on the first pass, when the records are read, and printed on the second pass. You cannot use aggregates in a loop, such as FOR or WHILE, where the WHERE clause changes dynamically.

The MIN( ) and MAX( ) Functions

These evaluate as the *minimum* value and *maximum* value (respectively) of the expression among all records, or among records qualified by the optional WHERE clause or GROUP keyword. For character data, *greater than* means "after" in the ASCII collating sequence, where a> A> 1, and *less than* means "before" in the ASCII sequence, where 1< A< a. For DATE or DATETIME data, *greater than* means "later" and *less than* means "earlier" in time. See Appendix G for a listing of the ASCII collating sequence.

The AVG( ) and SUM( ) Functions

These evaluate as the *average* (that is, the *arithmetic mean* value) and the total (respectively) of the expression among all records, or among records qualified by the optional WHERE clause or GROUP keyword. The expression (in parentheses) that AVG( ) or SUM( ) evaluates must return a *4GL* variable or expression of a *number* or INTERVAL data type.

The COUNT (*) and PERCENT (*) Functions

These are evaluated, respectively, as the *total number* of records qualified by the optional WHERE clause, and as a *percentage* of the total number of records in the report. You must include the (*) symbols. Like the other report aggregates, PERCENT(*) and COUNT(*) cannot be used within an expression.

The following fragment of a REPORT routine uses the AFTER GROUP OF control block and GROUP keyword to form sets of records according to how many items are in each order. The last PRINT statement calculates the total price of each order, then adds a shipping charge, and prints the result.

```
AFTER GROUP OF number
  SKIP 1 LINE
  PRINT 4 SPACES, "Shipping charges for the order:  ",
    ship_charge USING "$$$$.&&"
  PRINT 4 SPACES,  "Count of small orders:  ",
    COUNT(*) WHERE total_price < 200.00 USING "##,###"
  SKIP 1 LINE
  PRINT 5 SPACES, "Total amount for the order:  ",
    ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$$.&&"
```

With no WHERE clause, GROUP SUM here combines *every* item in the group.

Differences Between the 4GL and SQL Aggregates

The *Informix Guide to SQL: Reference* describes the syntax of the SQL aggregate functions. The following are the major differences between the 4GL report aggregates and aggregate functions that Informix database engines support:

- Only 4GL report aggregates can use the PERCENT(*) or GROUP keywords.
- Only SQL aggregates can use ALL, DISTINCT, or UNIQUE as keywords.
- In 4GL reports, COUNT can only take an asterisk (*) symbol as its argument; but in SELECT or DELETE statements of SQL, the COUNT aggregate can also use a column name or an SQL expression as its argument.

Only SQL aggregate functions can use database column names as arguments, but this syntax difference is not of much practical importance. (Operands in the expressions that you specify as arguments for 4GL report aggregates can be program variables that contain values from database columns.)

References

LINENO, PAGENO, WORDWRAP

ARG_VAL( )

The ARG_VAL( ) function returns a specified argument from the command line that invoked the current 4GL application program. It can also return the *name* of the current program.

```
ARG_VAL ——— ( ordinal ) ———————————————————————|
```

ordinal is an integer expression that evaluates to a non-negative whole number no larger than the number of arguments of the program. (See the syntax of "Integer Expressions" on page 3-338.)

Usage

This function provides a mechanism for passing values to the 4GL program through the command line that invokes the program. You can design a 4GL program to expect or to allow arguments after the name of the program in the command line.

Use the ARG_VAL( ) function to retrieve individual arguments during program execution. (You can also use the NUM_ARGS( ) function to determine how many arguments follow the program name on the command line.)

If $1 \leq ordinal = n$, then ARG_VAL(*n*) returns the nth command-line argument as a character string. The value of *ordinal* must be between 0 and the value returned by NUM_ARGS ( ), the number of command-line arguments.

The expression ARG_VAL(0) returns the *name* of the 4GL application program.

Using ARG_VAL( ) with NUM_ARGS( )

The built-in ARG_VAL( ) and NUM_ARGS( ) functions can pass data to a compiled 4GL program from the command line that invoked the program.

For example, suppose that the 4GL program called **myprog** can accept one or more login names as command-line arguments. Then both of the following command lines include the same four arguments:

```
myprog.4gi joe bob sue les        (C compiler version)
fglgo myprog joe bob sue les (RDS version)
```

In either case, statements in the following program fragment use the ARG_VAL( ) function to store in an array of CHAR variables all the names that the user who invoked **myprog** entered as command-line arguments:

```
VARIABLE args ARRAY[8] OF CHAR(10),
       i    SMALLINT
. . .
FOR i = 1 TO NUM_ARGS()
    LET args[i] = ARG_VAL(i)
END FOR
```

After the command-line arguments listed above, the NUM_ARGS( ) function returns the value 4. Executing the LET statements in the FOR loop assigns the following values to elements of the **args** array:

Variable	Value
args[1]	joe
args[2]	bob
args[3]	sue
args[4]	les

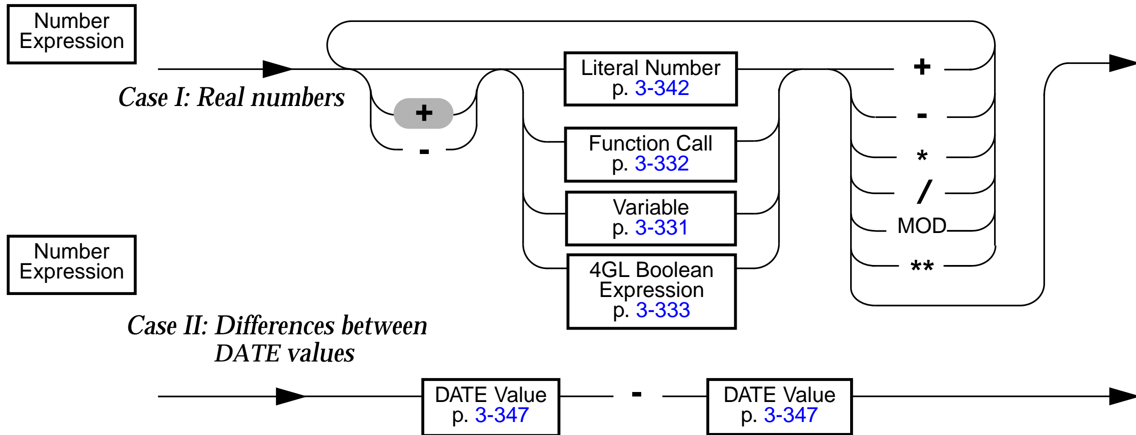
Reference

NUM_ARGS( )

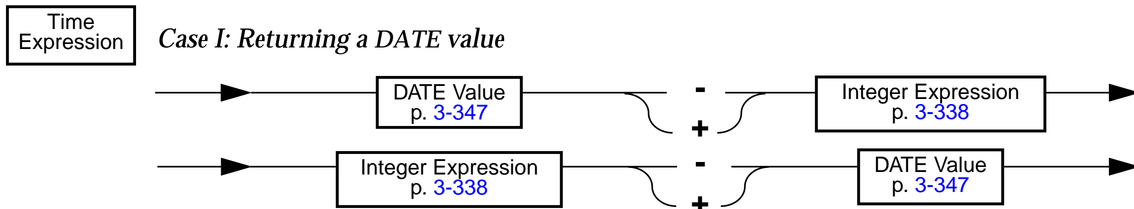
Arithmetic Operators

The 4GL *arithmetic operators* perform arithmetic operations on operands of number data types (and in some cases, of time data types).

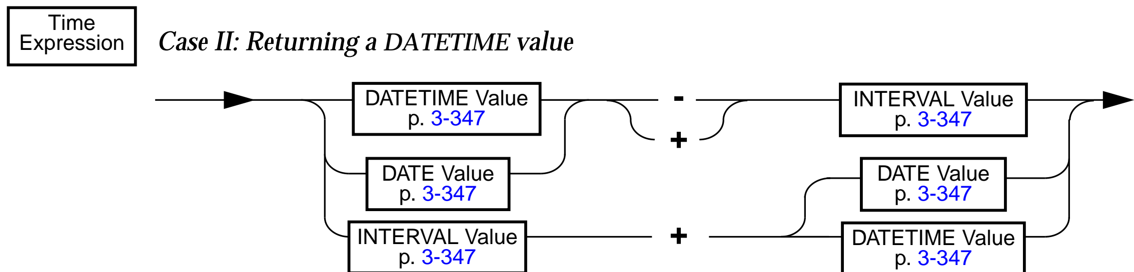
This is the syntax for arithmetic expressions that return a number value:



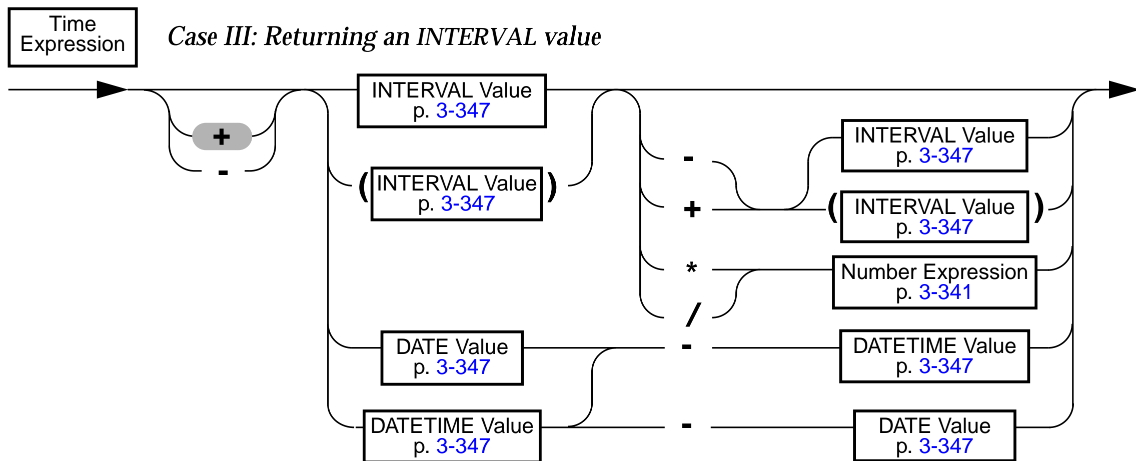
This is the syntax for arithmetic expressions that return a DATE value:



This is the syntax for arithmetic expressions that return a DATETIME value:



This is the syntax for arithmetic expressions that return an INTERVAL value:



Usage

If any component of an expression that includes an arithmetic operator is a NULL value, then the entire expression returns NULL, unless the NULL value is an operand of the IS NULL or IS NOT NULL operators.

Unary Arithmetic Operators

At the left of expressions that return a number or INTERVAL value, plus (+) and minus (–) symbols can appear as *unary operators* to specify the sign. For unsigned values, the default is positive (+). The number data types of 4GL are DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, and SMALLINT.

Unary plus (+) and minus (–) operators are recursive. Parentheses (`( )`) must separate the subtraction (–) operator from any immediately following unary minus sign, as in "`minuend -(-subtrahend)`," unless you want *4GL* to interpret the "--" symbols as a comment indicator.

Binary Arithmetic Operators

Six *binary arithmetic operators* can appear in number expressions. As the diagrams above indicate, four of these (`*`, `/`, `+`, and –) also can appear in time (DATE, DATETIME, and INTERVAL) expressions. The MOD and exponentiation (`**`) operators accept some DATE values as operands, but such expressions may return values that are very difficult to interpret.

Operator Symbol	Operator Name	Name of Result	Precedence
**	exponentiation	power	12
`mod`	modulus	remainder	12
*	multiplication	product	11
/	division	quotient	11
+	addition	sum	10
-	subtraction	difference	10

Note: 4GL *performs calculations with binary arithmetic operators of number data types after automatically converting both operands to* DECIMAL *values.*

The following table shows the precedence (**P**) and data types of operands and of returned values for both unary and binary arithmetic operators. Time operands not listed here produce either errors or meaningless results.

P	Expression	Left (= x)	Right (= y)	Returned Value
13	+ **y**		**Number** *or* **INTERVAL**	*Same as* **y**
	- **y**		**Number** *or* **INTERVAL**	*Same as* **y**
12	**x** * * **y**	**Number**	**INT** *or* **SMALLINT**	*Same as* **y**
	x MOD **y**	**INT** *or* **SMALLINT**	**INT** *or* **SMALLINT**	*Same as* **y**
11	**x** * **y**	**Number** *or* **INTERVAL**	**Number**	*Same as* **x**
	x / **y**	**Number** *or* **INTERVAL**	**Number**	*Same as* **x**
10	**x** + **y**	**Number**	**Number**	**Number**
	x + **y**	**INT** *or* **SMALLINT**	**DATE**	**DATE**
	x + **y**	**DATE**	**INT** *or* **SMALLINT**	**DATE**
	x + **y**	**DATE** *or* **DATETIME**	**INTERVAL**	**DATETIME**
	x + **y**	**INTERVAL**	**DATE** *or* **DATETIME**	**DATETIME**
	x + **y**	**INTERVAL**	**INTERVAL**	**INTERVAL**
	x - **y**	**Number**	**Number**	**Number**
	x - **y**	**INT** *or* **SMALLINT**	**DATE**	**DATE**
	x - **y**	**DATE**	**INT** *or* **SMALLINT**	**DATE**
	x - **y**	**DATE** *or* **DATETIME**	**INTERVAL**	**DATETIME**
	x - **y**	**DATE** *or* **DATETIME**	**DATETIME**	**INTERVAL**
	x - **y**	**DATETIME**	**DATE**	**INTERVAL**
	x - **y**	**INTERVAL**	**INTERVAL**	**INTERVAL**
	x - **y**	**DATE**	**DATE**	**INT**

Note: *Page 3-328* *lists the precedence of all* 4GL *operators; the operators that are not listed on that page have a precedence of 1. These precedence* (**P**) *values are ordinal numbers to show relative ranks; future releases of* INFORMIX-4GL *may introduce additional operators, at which time these precedence values may change.*

Variables used as arithmetic operands can only be of simple data types (page 3-294). Structured (ARRAY or RECORD) or blob (BYTE or TEXT) values are not valid operands. The range of returned values is that of the returned data type.

If a 4GL Boolean expression is an arithmetic operand, 4GL evaluates it and then converts it to an integer by these rules: TRUE = 1 and FALSE = 0. If the Boolean expression returns a NULL value, so does the arithmetic expression.

Restrictions on Time Operands

If the first operand of an arithmetic expression includes the UNITS operator (page 4-89), then you must enclose that operand within parentheses.

An expression cannot combine an INTERVAL value of precision in the range YEAR to MONTH with another in the DAY to FRACTION range, nor combine an INTERVAL value with a DATETIME or DATE value that has different qualifiers. You must explicitly use the EXTEND operator (page 4-53) to change the DATE or DATETIME precision to match that of the INTERVAL operand.

You can use DATE operands in addition and subtraction, but not the sum of two DATE values. All the other binary arithmetic operators (page 4-19) also accept DATE operands, equivalent to the count of days since December 31, 1899; but the values returned (except from a DATE expression as the left-hand MOD operand) are meaningless in most applications.

Note: *DATE and DATETIME values have no true zero point. Such values logically support addition, subtraction, and the relational operators (page 4-32), but division, multiplication, and exponentiation are logically undefined for these data types.*

Exponentiation (**) Operator

The exponentiation (**) operator returns a value calculated by raising the left-hand operand to a power corresponding to the integer part of the right-hand operand. This right-hand operand cannot have a negative value.

An expression specifying the right-hand MOD operand cannot include the exponentiation (**) operator.

Before conversion to DECIMAL for evaluation, 4GL converts the right-hand operand of the exponentiation (**) operator to an INTEGER value. Any fractional part is discarded.

Modulus (MOD) Operator

The modulus (MOD) operator returns the remainder from integer division when the integer part of the left-hand operator is divided by the integer part of the right-hand operator. For example, if y = 7.76 and z = 2.95, then

```
LET x = y MOD z
```

assigns to **x** the value 1, the integer part of the remainder of 7 divided by 2.

In 4GL programs, MOD is a reserved word. Do not use it as a *4GL* identifier.

An expression specifying the right-hand MOD operand cannot include the exponentiation (**) nor modulus (MOD) operators, and cannot be zero.

Before conversion to DECIMAL for evaluation, 4GL converts any operand of MOD that is not of the INTEGER or SMALLINT data types to an INTEGER value by truncation. Any fractional part is discarded.

Multiplication (*) and Division (/) Operators

The multiplication (*) operator returns the scalar product of its left-hand and right-hand operands.

The division operator returns the quotient of its left-hand operand divided by its right-hand operand. An error is returned if the right-hand operand (the divisor) evaluates to zero.

If both operands of the division (/) operator have INT or SMALLINT data types, then *4GL* discards any fractional portion of the quotient.

For multiplication and division, if the left-hand operand has an INTERVAL value, the result is an INTERVAL value of the same precision. (The right-hand operand must be an expression that returns a number data type.)

Addition (-) and Subtraction (+) Operators

The addition (+) and subtraction (-) operators return the algebraic sum and difference, respectively, between their left- and right-hand operands.

Do not write expressions that specify the sum (+) of two DATE or DATETIME values, nor a difference (–) whose second operand is a DATE or DATETIME value, and whose first operand is an INTERVAL value.

The difference between two DATETIME values (or a DATETIME and a DATE value, but *not* two DATE values) is an INTERVAL value. If the operands have different qualifiers, the result has the qualifier of the first operand.

The difference between two DATE values is an INTEGER value, representing the positive or negative number of *days* between the two calendar dates. You must explicitly apply the UNITS DAY operator to the difference between DATE values, if you wish to store the result as an INTERVAL value.

References

Aggregate Report Functions, Boolean Operators, EXTEND, UNITS

ARR_COUNT( )

The ARR_COUNT( ) function returns a positive whole number, typically representing how many records were entered in a program array during or after execution of the INPUT ARRAY statement.

ARR_COUNT( ) ——————————————————————————|

Usage

You can use ARR_COUNT( ) to determine the number of program records that are currently stored in a program array. In typical 4GL applications, these records correspond to values from the active set of retrieved database rows from the most recent query. By first calling the SET_COUNT( ) function (page 4-80), you can set an upper limit on the value that ARR_COUNT( ) returns.

ARR_COUNT( ) returns a positive integer, corresponding to the index of the furthest record within the program array that the screen cursor accessed. Not all the rows "counted" by ARR_COUNT( ) necessarily contain data (for example, if the user presses the Down arrow key more times than there are rows of data). If SET_COUNT( ) was explicitly called, ARR_COUNT( ) returns the greater of these two values: the argument of SET_COUNT( ), or the highest value attained by the array index.

The **insert_items( )** function in the following example uses the value returned by ARR_COUNT( ) to set the upper limit in a FOR statement:

```
FUNCTION insert_items()
   DEFINE counter SMALLINT
   FOR counter = 1 TO ARR_COUNT()
      INSERT INTO items
         VALUES (p_items[counter].item_num,
            p_orders.order_num,
            p_items[counter].stock_num,
            p_items[counter].manu_code,
            p_items[counter].quantity,
            p_items[counter].total_price)
   END FOR
END FUNCTION
```

The following example makes use of ARR_COUNT( ) and the related built-in functions ARR_ CURR( ) and SCR_LINE( ) to assign values to variables within the BEFORE ROW clause of an INPUT ARRAY WITHOUT DEFAULTS statement.

By calling these functions in BEFORE ROW, the respective variables are evaluated each time the cursor moves to a new line and are available within other clauses of the INPUT ARRAY statement.

```
INPUT ARRAY ga_manuf WITHOUT DEFAULTS FROM sa_manuf.*
   BEFORE ROW
      LET curr_pa = ARR_CURR()
      LET curr_sa SCR_LINE()
      LET total_pa = ARR_COUNT()
```

It is then possible, for example, in a later statement within INPUT ARRAY, to have a statement such as the following, which tests whether the cursor is at the last position in the screen array:

```
   IF curr_pa <> total_pa THEN       ...
```

References

ARR_CURR( ), SCR_LINE( ), SET_COUNT( )

ARR_CURR( )

During or immediately after the INPUT ARRAY or DISPLAY ARRAY statement the ARR_CURR( ) function returns the number of the program record within the program array that is displayed in the current line of a screen array.

ARR_CURR( ) ——————————————————————————┤

Usage

The current line of a screen array is the line that displays the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The ARR_CURR( ) function returns an integer value. The first row of the program array and the first line (that is, topmost) of the screen array are both numbered 1. The built-in functions ARR_CURR( ) and SCR_LINE( ) can return different values if the program array is larger than the screen array.

You can pass ARR_CURR( ) as an argument when you call a function. In this way the function receives as its argument the current record of whatever array is referenced in the INPUT ARRAY or DISPLAY ARRAY statement.

The ARR_CURR( ) function can be used to force a FOR loop to begin beyond the first line of an array by setting a variable to ARR_CURR( ) and then using that variable as the starting value for the FOR loop.

The following program segment tests the user input for duplication of what should be a unique column. If the field duplicates an existing item, the program instructs the user to try again.

```
INPUT ARRAY ga_manufact FROM sa_manufact.*
    AFTER FIELD manu_code
        IF pk_check(ARR_CURR())  THEN
            ERROR "This code already exists. Re-enter",
            " or press F2 to delete this entry."
            NEXT FIELD manu_code
        END IF
END INPUT
```

In this example, the value returned by ARR_CURR( ) is then passed to function **pk_check( )**, where it is stored in the local variable **el_pa**, serving as an index to the global array **ga_manufact**:

```
FUNCTION pk_check(el_pa) --verifies primary key
   DEFINE el_pa, manu_count INT
   SELECT COUNT(*) INTO manu_count
       FROM manufact
       WHERE manufact.manu_code = ga_manufact[el_pa].manucode
   IF manu_count >= 1
       THEN RETURN TRUE
       ELSE RETURN FALSE
   END IF
END FUNCTION
```

The ARR_CURR( ) function is frequently used in conjunction with a DISPLAY ARRAY statement in pop-up windows to return the user's selection.

The following example allows users to choose supplier codes for shoes in an order form. The user chooses among eight possibilities. The choice is returned and displayed on a form. The variables **pa_supplier** and **elem_pa** are locally defined. The variable **gr_shoes** is a global record associated with an INPUT statement.

```
   OPEN WINDOW w_supplier AT 3,50
      WITH FORM "f_popscode"
      ATTRIBUTE  (BORDER, REVERSE)
   DISPLAY "Press ESC to select." AT 1,1
   DISPLAY "Use arrow keys to move." at 2,1
   CALL SET_COUNT(8)
   DISPLAY ARRAY pa_supplier TO sa_supplier.*
      LET elem_pa = ARR_CURR()
      LET gr_shoes.supply_code = pa_supplier[elem_pa].s_code
   CLOSE WINDOW w_supplier
   DISPLAY BY NAME gr_shoes.supply_code
```

References

ARR_COUNT( ), SCR_LINE( )

ASCII

The ASCII operator converts an integer operand into its corresponding ASCII character.

```
ASCII number ─────────────────────────────────────────┤
```

number is an integer expression (page 3-338) that returns a positive whole number within the range of ASCII values.

Usage

You can use the ASCII operator to evaluate an integer to a single character. This operator is especially useful if you need to display CONTROL characters.

The following DISPLAY statement rings the terminal bell (ASCII value of 7).

```
DEFINE bell CHAR(1)
LET bell = ASCII 7
DISPLAY bell
```

The next REPORT program block fragments show how to implement special printer or terminal functions. They assume that, when the printer receives the sequence of ASCII characters 9, 11, and 1, it will start printing in red, and when it receives 9, 11, and 0, it will revert to black printing. The values used in the example are hypothetical; refer to your printer or terminal manual for information on your printer or terminal.

```
FORMAT
   FIRST PAGE HEADER
      LET red_on = ASCII 9, ASCII 11, ASCII 1
      LET red_off = ASCII 9, ASCII 11, ASCII 0
   ON EVERY ROW
   ...
      PRINT red_on,
            "Your bill is overdue.", red_off
```

Caution: 4GL *cannot distinguish printable and non-printable* ASCII *characters. Be sure to account for the non-printing characters when using the* COLUMN *operator to format your page. Since various devices differ in outputting spaces with control characters, you may have to use trial and error to line up columns when you output control characters.*

The ASCII Operator in PRINT Statements

To print a NULL character in a report, call the ASCII operator with 0 in a PRINT statement. For example, the following statement prints the NULL character:

```
PRINT ASCII 0
```

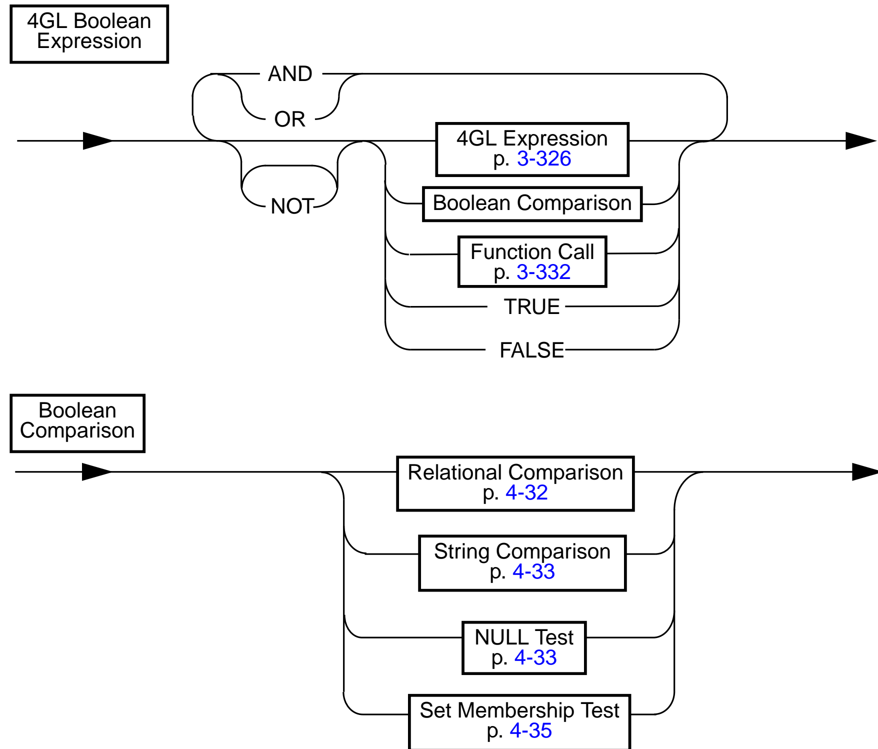
ASCII 0 only displays the NULL character within the PRINT statement. If you specify ASCII 0 in other contexts, it returns a blank space.

References

FGL_KEYVAL( ), FGL_LASTKEY( )

Boolean Operators

A *4GL Boolean operator* returns TRUE (= 1), FALSE (= 0) or NULL. They closely resemble the set of SQL Boolean operators, but some are not identical.



Usage

The Boolean operators include the *logical operators* AND, OR, and NOT, and operators for *Boolean comparisons*, as described in the sections that follow.

Logical Operators

The *logical operators* AND, OR, and NOT combine Boolean values into a single *4GL* Boolean expression. AND, OR, and NOT produce the following results (where the symbol `T` means TRUE, `F` means FALSE, and `?` means NULL):

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

When one or both arguments of a logical operator are NULL, the result can in some cases also be NULL. For example, if **var1** = 0 and **var2** = NULL, then

```
LET x = var1 OR var2
```

assigns to the variable **x** a NULL value. The NOT operator is recursive.

Boolean Comparisons

Boolean comparisons can use the following Boolean operators:

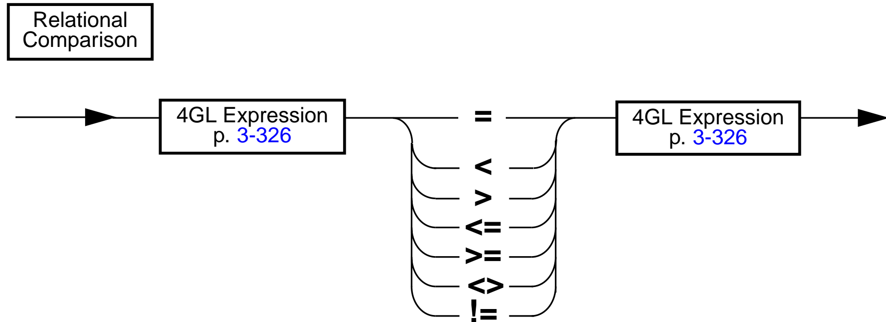
- *Relational operators* to test for equality or inequality
- IS NULL to test for NULL values
- LIKE or MATCHES to compare *character strings*
- IN or BETWEEN . . . AND to test for *set membership* or for *range.*

Note: *The IN and BETWEEN . . . AND operators are valid only in SQL statements. They cause a compile-time error if you include them in other 4GL statements. They are included here, however, because they are valid in the WHERE clause of a form specification file that includes the COLOR attribute (page 5-32).*

Boolean expressions in the CASE, IF, or WHILE statements (or in the WHERE clause of a COLOR attribute specification) return FALSE if any element of the comparison is NULL, unless it is the operand of the IS NULL operator.

Relational Operators

This is the syntax for relational comparisons in *4GL* Boolean expressions:



Boolean expressions in *4GL* statements can use these *relational operators* (=, ==, <, >, <=, >=, <>, or !=, as defined on page 4-37) to compare operands. For example, each of these comparisons returns TRUE or FALSE:

Expression	Value
(2+5)* 3 = 18	FALSE
14 <= 16	TRUE
"James" = "Jones"	FALSE

Use a NULL test (page 4-33) if you want to detect and exclude NULL values from Boolean comparisons. In this CASE statement fragment, the value of the comparison is NULL if the value of **salary** or of **last_raise** is NULL:

```
WHEN salary * last_raise < 25000
```

For character expressions, the result depends on the position of the initial character of each operand within the ASCII collating sequence.

For number expressions, the result of a relational comparison reflects the position of the two operands on the real line.

Relational comparisons of time expression operands follow these rules:

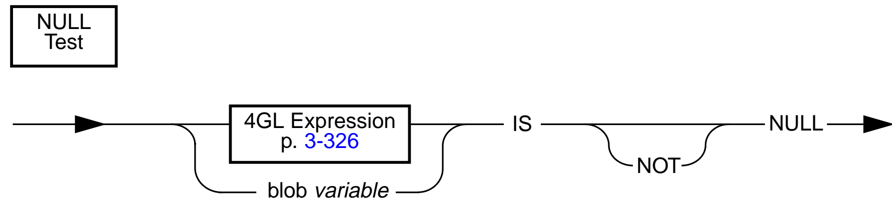
- Comparison `x < y` is TRUE when `x` is a *briefer* INTERVAL span than `y`, or when `x` is an *earlier* DATE or DATETIME value than `y`.
- Comparison `x > y` is TRUE when `x` is a *longer* INTERVAL span than `y`, or when `x` is a *later* DATE or DATETIME value than `y`.
- You cannot mix INTERVAL operands with DATE or DATETIME operands, but you can compare DATE and DATETIME expressions with each other.

The NULL Test

If any operand of a *4GL* Boolean comparison or is NULL, then the value of the comparison is FALSE (rather than NULL), unless the IS NULL keywords are also included in the expression. Applying the NOT operator to a NULL value does not change its FALSE evaluation.

If you need to process expressions with NULL values in a different way from other values, you can use the IS NULL keywords to test for NULL value

NULL Test



blob variable is the name of a variable of the BYTE or TEXT data type.

Without the NOT keyword, the comparison returns TRUE if the operand has a NULL value. If you include the NOT keyword, the comparison returns TRUE if the value of the operand is not NULL. Otherwise, it returns FALSE.

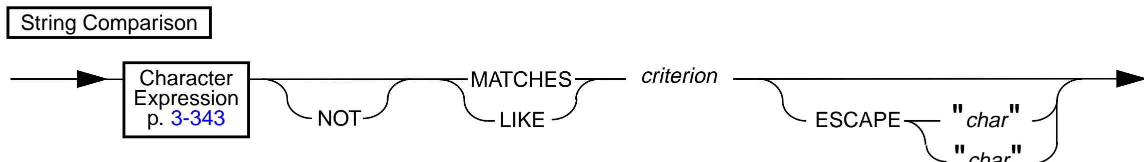
Note: *The NULL test (like the WORDWRAP string operator with TEXT variables) is an exception to the general rule that variables of the BYTE or TEXT data types cannot appear as operands in 4GL expressions.*

The LIKE and MATCHES Operators

The LIKE or MATCHES operators test whether a character value matches a quoted string that can include *wildcard characters.* If an operand has a NULL value, then the entire string comparison returns NULL. Use a NULL test (as in the previous section) if you want to detect and exclude NULL values.

You can use the following syntax to compare character strings:

String Comparison



char is a single ASCII character, enclosed between a pair of single (") or double (") quotation marks, to specify an escape symbol.

criterion is a character expression (3-343). The string that it returns can include literal characters, wildcards, and other symbols.

MATCHES and LIKE support different wildcards. If you use MATCHES, you can include the following wildcard characters in the right-hand operand:

Symbol	Effect
`*`	An asterisk (`*`) matches any string of zero or more characters.
`?`	A question mark (`?`) matches any single character.
`[ ]`	Square brackets (`[ ]`) match any of the enclosed characters.
`-`	A hyphen (`-`) between characters in brackets means a range in the ASCII collating sequence. For example, `[a-z]` matches any lowercase letter.
`^`	An initial caret (`^`) in the brackets matches any character that is not listed. For example, `[^abc]` matches any character except `a`, `b`, or `c`.
`\`	Backslash (`\`) causes **4GL** to treat the next character as a literal character, even if it is one of the special symbols in this list. For example, you can match `*` or `?` by `*` or `\?` in the string.

The following WHERE clause tests the contents of character field **field007** for the string `ten`. Here the `*` wildcards specify that the comparison is TRUE if `ten` is found alone or in a longer string, such as `often` or `tennis shoe`:

```
COLOR = RED WHERE field007 MATCHES "*ten*"
```

If you use the keyword LIKE to compare strings, then the wildcard symbols of MATCHES have no special significance, but you can use the following wildcard characters of LIKE within the right-hand quoted string:

Symbol	Effect
%	A percent sign (%) matches zero or more characters.
_	An underscore (_) matches any single character.
\	A backslash (\ ) causes **4GL** to treat the next character as a literal (so you can match % or _ by \% or _).

The next example tests for the string `ten` in the character variable **string**, either alone or in a longer string:

```
IF string LIKE "%ten%"
```

The next example tests whether a substring of a character variable (or else an element of a two-dimensional array) contains an underscore symbol. The backslash is necessary, because underscore is a wildcard symbol with LIKE.

```
IF horray[3,8] LIKE "%\_%" WHERE >> out.a
```

You can replace backslash as the literal symbol. If you include an ESCAPE *char* clause in a LIKE or MATCHES specification, then INFORMIX-4GL interprets the next character that follows *char* as a literal in the preceding character expression, even if that character corresponds to a special symbol of the LIKE or MATCHES keyword. The double quote (**"**) symbol cannot be *char*.

For example, if you specify ESCAPE `z`, the characters `z_` and `z?` in a string stand for the literal character _ and `?`, rather than wildcards. Similarly, characters `z%` and `z*` stand for the characters `%` and `*`. Finally, the characters `zz` in the string stand for the single character `z`. The following expression is TRUE if the variable **company** does not include the underscore character:

```
NOT company LIKE "%z_%" ESCAPE "z"
```

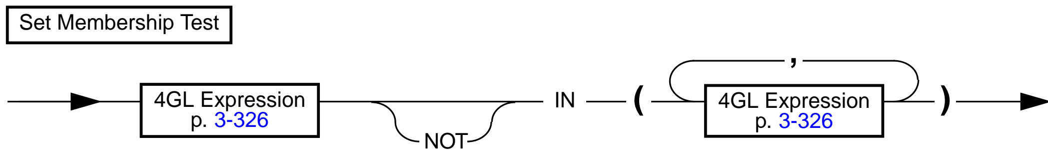
Set Membership and Range Tests

The BETWEEN . . . AND and IN( ) operators that test for set membership or range are supported in three contexts:

- In SQL statements
- In the WHERE clause of the COLOR attribute in 4GL form specifications
- In the **condition** column of the **syscolatt** table

They are not valid in 4GL statements that are not also SQL statements.

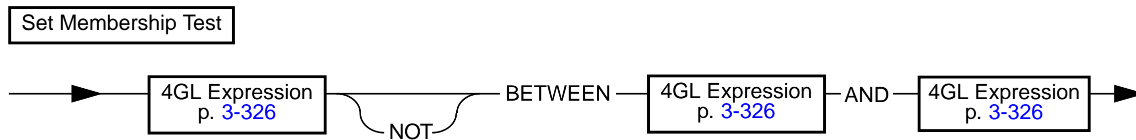
This is the syntax for using the IN( ) operator to test for set membership:



If you omit the NOT keyword, this returns TRUE if any expression in the list (within parentheses) at the right matches the expression on the left.

If you include the NOT keyword, the test evaluates as FALSE if no expression in the list matches the expression on the left.

This is the syntax for the BETWEEN . . . AND operators to test whether a value is included within a specified range (or an inclusive interval on the real line):



Operands must return compatible data types. Values returned by the second (O2) and third (O3) operands that define the range must follow these rules:

- For number or INTERVAL values, O2 must be less than or equal to O3.
- For DATE and DATETIME values, O2 must be no later than O3.

- For character strings, O2 must be earlier than O3 in the ASCII collating sequence. (This is listed in Appendix G.)

If you omit the NOT keyword, this test evaluates as TRUE if the first operand has a value not less than the second operand, nor greater than the third. If you include the NOT keyword, the test evaluates as FALSE if the first operand has a value outside the specified range.

Data Type Compatibility

You may get unexpected results if you use relational operators with expressions of dissimilar data types. In general, you can compare numbers with numbers, character strings with strings, and time values with time values.

If a ***time expression*** operand of a *4GL* Boolean expression is of the INTERVAL data type, then any other time expression that is compared to it by a relational operator must also be an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value). For additional information about data type compatibility in expressions, see the section "Data Type Conversion" on page 3-319.

Evaluating 4GL Boolean Expressions

In contexts where a *4GL* Boolean expression is expected, INFORMIX-4GL applies the following rules after it evaluates the expression:

- If the value is a non-zero real number (or a character string representing one) or a non-zero INTERVAL, or any DATE or DATETIME value, or a TRUE value returned by a built-in Boolean function or operator like INFIELD( ), or the integer constant TRUE, then the *4GL* Boolean value is TRUE.
- If the value is NULL, but the expression is the operand of the IS NULL keywords, then the value of the *4GL* Boolean expression is TRUE.
- If the value is NULL, and the expression is not an operand of a NULL test, nor an element in any Boolean comparison or conditional statement of *4GL* (IF, CASE, WHILE), then the expression returns NULL.
- Otherwise, the *4GL* Boolean expression is evaluated as FALSE.

If a Boolean expression has several operators, they are processed according to their precedence. Operators that have the same precedence are processed from left to right. For the Boolean operators of 4GL, the following table lists their precedence (**P**) and summarizes the data types of their operands.

P	Description	Expression	Left (= x)	Right (= y)	Returns
9	string comparison	**x** LIKE **y**	**Character**	**Character**	**Boolean**
	string comparison	**x** MATCHES **y**	**Character**	**Character**	**Boolean**
8	*test for*: less than	**x** < **y**	Any simple data type	*Same as* **x**	**Boolean**
	less than *or* equal to	**x** <= **y**	Any simple data type	*Same as* **x**	**Boolean**
	equal to	**x** = **y** *or* **x** == **y**	Any simple data type	*Same as* **x**	**Boolean**
	greater than *or* equal to	**x** >= **y**	Any simple data type	*Same as* **x**	**Boolean**
	greater than	**x** > **y**	Any simple data type	*Same as* **x**	**Boolean**
	not equal to	**x** != **y** *or* **x** <> **y**	Any simple data type	*Same as* **x**	**Boolean**
7	*test for*: set membership	**x** IN (**y**)	Any	Any	**Boolean**
6	*test for*: range	**x** BETWEEN **y** AND **z**	Any	*Same as* **x**	**Boolean**
5	*test for*: NULL	**x** IS NULL	Any		**Boolean**
	test for: NULL	**x** IS NOT NULL	Any		**Boolean**
4	logical inverse	NOT **y**		**Boolean**	**Boolean**
3	logical intersection	**x** AND **y**	**Boolean**	**Boolean**	**Boolean**
2	logical union	**x** OR **y**	**Boolean**	**Boolean**	**Boolean**
1	test whether field edited	FIELD_TOUCHED(**y**)		*Field name*	**Boolean**
	test for current field	INFIELD(**y**)		*Field name*	**Boolean**

Note: *Page 3-328 lists the precedence of all* 4GL *operators. These relative precedence* (**P**) *values are ordinal numbers; future releases of* INFORMIX-4GL *may introduce additional operators, at which time these precedence values may change.*

Besides the Boolean operators that are listed in this table, the built-in 4GL operators FIELD_TOUCHED( ) and INFIELD( ) also return Boolean values. Their precedence is lower (**P** = 1) than that of the OR operator. They can use the name of a field in the current form as their operand. Both the INFIELD( ) and FIELD_TOUCHED( ) operators are described later in this chapter.

References

FIELD_TOUCHED( ), INFIELD( )

CLIPPED

The CLIPPED operator takes a character operand and returns the same character value, but without any trailing blank spaces (that is, ASCII 32).

Usage

The CLIPPED operator follows the character expression operand that you want to display without trailing blanks. Character expressions often have a data length less than their total size. The following DISPLAY statement, for example, would produce output that included 200 trailing blanks if CLIPPED were omitted, but displays only 22 characters when CLIPPED is included:

```
DEFINE string CHAR(222)
LET string = "Two hundred characters"
DISPLAY string CLIPPED
```

The CLIPPED operator can be useful in situations like the following:

- After a variable name in a DISPLAY, ERROR, LET, MESSAGE, or PROMPT statement, or in a PRINT statement of a REPORT program block.
- When concatenating several character expression into a single string.
- When comparing two or more character expressions and one or more of them is already clipped.

The CLIPPED operator can affect the value of a character variable within an expression. CLIPPED does not affect the value when it is stored in a variable (unless you are concatenating CLIPPED values together). For example, if CHAR variable **b** contains a string that is shorter than the declared length of CHAR variable **a**, the following LET statement pads **a** with trailing blanks, despite the CLIPPED operator:

```
LET a = b CLIPPED
```

Note: *To discard trailing blanks from a string of fewer than 255 characters, use a VARCHAR variable to store the string. For example, if CHAR variable* **b** *contains a string value no longer than the declared maximum size of VARCHAR variable* **a**, *then the following statement discards any trailing blanks from what it stored in* **a**:

```
LET a = b
```

The following program fragment is from a REPORT program block that prints mailing labels.

```
FORMAT
  ON EVERY ROW
    IF (city IS NOT NULL)
       AND (state IS NOT NULL) THEN
       PRINT fname CLIPPED, 1 SPACE, lname
       PRINT company
       PRINT address1
       IF (address2 IS NOT NULL) THEN
          PRINT address2
    END IF
       PRINT city CLIPPED, ", " , state,
       2 SPACES, zipcode
       SKIP TO TOP OF PAGE
  END IF
```

The following program fragment is from a report driver. Here CLIPPED is used to cosmetically improve the text of a MESSAGE statement that includes a filename stored in a character variable:

```
DEFINE file_name CHAR(60)
PROMPT " Enter drive, pathname, ",
       "and file name for Book Report:" FOR file_name
IF (file_name IS NULL) THEN
    LET file_name = "book.out"
END IF
MESSAGE "Printing Book Report to ", file_name CLIPPED,
        " --Please wait."
```

Reference

USING

COLUMN

COLUMN specifies the position in the current line of a report where output of the next value in a PRINT statement begins, or the position on the 4GL screen for the next value in a DISPLAY statement.

```
COLUMN left-offset ──────────────────────────────────────┤
```

left-offset is an integer expression (page 3-338) in a report, or else a literal integer (page 3-340) in a DISPLAY statement, to specify where the next character of output will appear.

Usage

The *left-offset* specifies a character position offset from the left margin of the 4GL screen or the currently executing 4GL report. In a report, this cannot be greater than the arithmetic difference (*right margin* – *left margin)* for explicit or default values in the OUTPUT section of the REPORT program block (page 6-9). See the description of integer expressions (page 3-338) for the syntax of 4GL expressions that return whole numbers.

Unless you use the keyword CLIPPED or USING, the PRINT statement (and the DISPLAY statement when no form is open) display 4GL variables with widths (including any sign) that depend on their declared data types:

Data Type	Default Display Width (in characters)
CHAR	The length from the data-type declaration
DATE	10
DATETIME	From 2 to 25, as implied in the data-type declaration
DECIMAL	(2 + *m*), for *m* the precision from the data-type declaration
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25, as implied in the data-type declaration
MONEY	(3 + *m*), for *m* the precision from the data-type declaration
SMALLFLOAT	14
SMALLINT	6
VARCHAR	The maximum length from the data-type declaration

In a REPORT program block, or in a DISPLAY statement that outputs data to the 4GL screen, you can use the COLUMN operator to control precisely the location of items within a line. This is often a requirement for the output of tabular information, and it is convenient for many other uses.

If the printing position in the current line is already beyond the specified *left-offset*, the COLUMN operator has no effect.

COLUMN in DISPLAY Statements

4GL calculates the left-offset from the first character position of the 4GL screen. In the following statements, for example,

```
DISPLAY "NUMBER", COLUMN 12, "NAME", COLUMN 35,
  "CITY", COLUMN 57, "ZIP", COLUMN 65, "PHONE"
DISPLAY ASCII 13, customer_num, COLUMN 12, fname CLIPPED,
  ASCII 32, lname CLIPPED, COLUMN 35, city CLIPPED,
  ", ", state, COLUMN 57, zipcode, COLUMN 65, phone
```

both the string NAME and the 4GL variable **fname** are each displayed with their first (left-most) character in the 12th character position on the 4GL screen. Output from each DISPLAY statement begins on a new line.

Note: *You cannot use* COLUMN *to send output to a screen form. Any* DISPLAY *statement that includes the* COLUMN *operator cannot also include the* AT, TO, BY NAME, *or* ATTRIBUTE *clause. When you include the* COLUMN *operator in a* DISPLAY *statement, you must specify a literal integer as the left-offset, rather than an integer expression.*

COLUMN in PRINT Statements

When you use the PRINT statement in the FORMAT section of a report, by default items are printed one following the other, separated by spaces. The COLUMN operator can override this default positioning. 4GL calculates the *left-offset* from the left margin that you set in the OUTPUT section. If no left margin is specified, the *left-offset* is counted from the left margin of the page. If the following PRINT statements (with COLUMN and SPACE specifications) were part of a report that sent output to the 4GL screen, the output would resemble that of the DISPLAY statements in the previous example:

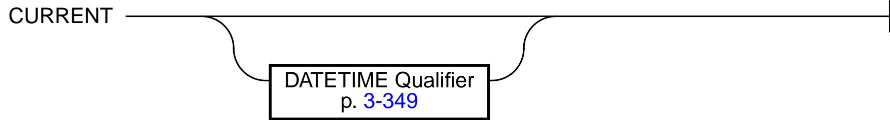
```
PAGE HEADER
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35,
     "CITY", COLUMN 57, "ZIP", COLUMN 65, "PHONE"
     SKIP 1 LINE
ON EVERY ROW
   PRINT customer_num, COLUMN 12, fname CLIPPED,
     1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED,
     ", ", state, COLUMN 57, zipcode, COLUMN 65, phone
```

References

ASCII, CLIPPED, SPACE, USING

CURRENT

The CURRENT operator returns the current *date* and *time-of-day* from the system clock as a DATETIME value of a specified or default precision.



Usage

The CURRENT operator reads the date and time from the system clock.

You can optionally specify the precision of the returned value by including a qualifier of the form *first* TO *last*, for *first* and *last* keywords from this list:

YEAR	MONTH	DAY	
HOUR	MINUTE	SECOND	FRACTION (*n*)

The *first* keyword must specify a DATETIME value that is the same as or more significant than the *last* keyword. That is, the following expression is valid:

```
CURRENT YEAR TO DAY
```

But this expression is not valid:

```
CURRENT MINUTE TO HOUR
```

If FRACTION is the last keyword, you can include a digit *n* in parentheses, to specify the number of digits of scale (up to 5) of the *seconds* value.

If no qualifier is specified, the default qualifier is YEAR TO FRACTION(3).

If CURRENT is executed more than once in a statement, identical values may be returned at each call. Similarly, the order in which the CURRENT operator is executed in a statement cannot be predicted. For this reason, you should not attempt to use this operator to mark the execution of the start, the end, or any specific point in a 4GL statement.

The CURRENT operator can be used both in SQL statements and in other 4GL statements. The following example is from an SQL statement:

```
SELECT prog_title FROM tv_programs
       WHERE air_date > CURRENT YEAR TO DAY
```

This is an example from a form specification file:

```
ATTRIBUTES  -- FORM4GL field

timestamp = FORMONLY.tmstmp TYPE DATETIME HOUR TO SECOND,
            DEFAULT = CURRENT HOUR TO SECOND;
```

The next example is from a report:

```
PAGE HEADER -- Report control block
        PRINT COLUMN 40, CURRENT MONTH TO MONTH,
              COLUMN 42, "/",
              COLUMN 43, CURRENT DAY TO DAY,
              COLUMN 45, "/",
              COLUMN 46, CURRENT YEAR TO YEAR
```

The last example would not produce the correct results if its execution spanned midnight.

References

DATE, EXTEND( ), TIME, TODAY

DATE

The DATE operator returns a character representation of the current date.

DATE ──┤

Usage

The DATE operator reads the system clock, and displays the current date in this format:

weekday month day year

as a character string, where:

weekday	is a 3-character abbreviation of the name of the day of the week.
month	is a 3-character abbreviation of the name of the month.
day	is a 2-digit representation of the day of the month.
year	is a 4-digit representation of the year.

The following example uses the DATE operator to display the current date:

```
VARIABLE p_date CHAR(15)
LET p_date = DATE
. . .
DISPLAY "Today is ", p_date AT 5,14
```

On Sunday, December 5th, 1993, this example would display the string:

```
Today is Sun Dec  5 1993
```

The effect of the DATE operator is sensitive to the time of execution and to the accuracy of the system clock.

An alternative way to format a DATE value as a character string is to use the FORMAT field attribute in a screen form (page 5-42), or to use the USING operator (page 4-94).

NLS

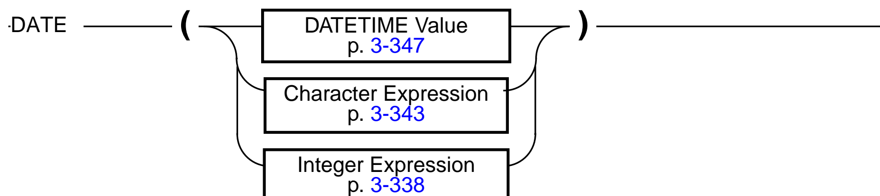
4GL can display language-specific month name and day name abbreviations. This requires the installation of message files in a subdirectory of **$INFORMIXDIR/msg**, and subsequent reference to that subdirectory by way of the environment variable DBLANG. For example, the *weekday* portion of the date string in a Spanish locale translates the day Saturday into the day name abbreviation Sab, which stands for Sabado (the Spanish word for Saturday). For more information on NLS, see Appendix E.

References

CURRENT, DATE( ), TIME, TODAY, USING

DATE( )

The DATE( ) operator can take a CHAR, VARCHAR, DATETIME, INTEGER, or SMALLINT argument and return the corresponding DATE value.



Usage

The DATE( ) operator is useful for data type conversion to a DATE value, or to a count of days since the beginning of the last year of the 19th century:

- Convert a properly formatted character string representation of a *numeric date* (page 3-349) to a DATE value. (The default format is "*mm*/*dd*/*yy*" but the DBDATE environment variable can change this default.)
- Converting a DATETIME value to a negative or positive integer. The returned integer corresponds to the number of days between the specified date and December 31, 1899.
- Obtaining a DATE value from a negative or positive integer, measuring the number of days between the specified date and December 31, 1899.

The following program fragment illustrates uses of the DATE( ) operator:

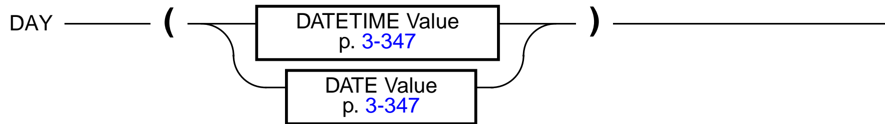
```
DEFINE d DATE
DEFINE dt DATETIME YEAR TO DAY
    LET d = DATE (" 11/20/99 ") -- this requires the default DATE format
    LET d = DATE (" 1999-11-02 ")  -- this requires that DBDATE be set to Y4MD-
    LET d = DATE (" 02:99:11 ")    -- this requires that DBDATE be set to DY2M:
LET d = DATE(d)  -- The operand can be a DATE variable, as here,
-- or the integer number of days since the last day of the year 1899
LET d = DATE (0)     -- result: 12/31/1899
LET d = DATE (34000) -- result: 2/1/1993
-- Or the operand can be a DATETIME type
LET dt = CURRENT
LET d = DATE (dt)     -- result is today's date
LET d = DATE(CURRENT) -- same result as previous
```

References

CURRENT, DATE, MDY( ), TODAY, UNITS

DAY( )

The DAY( ) operator returns a positive integer, corresponding to the day portion of the value of its DATE or DATETIME operand.



Usage

The DAY( ) operator can extract an integer value for the day of the month from a DATETIME or DATE operand. This feature is helpful in some applications, because INTEGER values are easier than DATETIME or DATE values to manipulate with arithmetic operators.

The following program fragment extracts the day of the month from a DATETIME literal:

```
DEFINE d_var    INTEGER,
       date_var DATETIME YEAR TO SECOND
LET date_var =  DATETIME (89-12-09 18:47:32) YEAR TO SECOND
LET d_var = DAY(date_var)
DISPLAY "The day of the month is: ", d_var USING "##"
```

References

MONTH( ), WEEKDAY( ), YEAR( )

DOWNSHIFT( )

The DOWNSHIFT( ) function returns a string value in which all uppercase characters in its argument are converted to lowercase.

DOWNSHIFT —— (Character Expression p. 3-343) ——|

Usage

The DOWNSHIFT( ) function is typically called to regularize character data. You might use it, for example, to prevent the state abbreviation "TX," "Tx," or "tx" from resulting in different values, if these were logically equivalent in the context of your application.

Non-alphabetic or lowercase characters are not altered by DOWNSHIFT( ). The maximum data length of the returned character string value is 511 bytes.

You can use the DOWNSHIFT( ) function in an expression (where such usage is allowed), or you can assign the value returned by the function to a variable.

In the following example, suppose that the CHAR value GEAR_4 is stored in the program variable **p_string**. The following statement takes the value of the expression DOWNSHIFT(**p_string**), namely gear_4, and assigns it to another CHAR variable called **d_str**:

```
LET d_str = DOWNSHIFT(p_string)
```

See also the DOWNSHIFT field attribute in Chapter 5.

NLS

When NLS is active, the results of conversion between uppercase and lowercase are appropriate to the national language in use, as defined by the LC_CTYPE environment variable.

Reference

UPSHIFT( )

ERR_GET( )

The ERR_GET( ) function returns a character string containing the text of the 4GL or SQL error message whose number you specify as its argument.

ERR_GET —— ([Integer Expression p. 3-338]) ——|

Usage

This is a possible sequence of steps for logging system error messages:

1. Use a WHENEVER ERROR CONTINUE compiler directive (page 3-281).
2. Call STARTLOG( ) to open or create an error log file (page 4-83).
3. Test the value of the global **status** variable to see if it is less than zero.
4. If **status** is negative, call ERR_GET( ) to retrieve the error text.
5. Call ERRORLOG( ) to make an entry into the error log file (page 4-51).

***Note:** If WHENEVER ERROR CONTINUE is not in effect, then the STARTLOG( ) function (step 2 above) automatically records the error text in the default error record, and your program does not need to invoke ERR_GET( ) explicitly.*

ERR_GET( ) is most useful when you are developing a program. The message that it returns is probably not helpful to the user of your 4GL application.

The argument of ERR_GET( ) is typically the value of the **status** variable, which is affected by both SQL and 4GL errors, or else a member of the global **SQLCA.SQLCODE** record. See "Exception Handling" on page 2-23.

The LET statement in the following program fragment assigns the text of a 4GL error message to **errtext**, a CHAR variable:

```
LET op_status = STATUS
IF op_status < 0 THEN LET errtext = ERR_GET(op_status)
END IF
```

(Here the value of **status** is first assigned to a variable, **op_status**, rather than testing ERR_GET(**status**) directly. Otherwise, the value of **status** normally would be zero, reflecting the success of the ERR_GET(**status**) function call.)

References

ERR_PRINT( ), ERR_QUIT( ), ERRORLOG( ), STARTLOG( )

ERR_PRINT( )

The ERR_PRINT( ) function displays on the Error line the text of an SQL or 4GL error message, corresponding to a negative integer argument.

```
CALL ERR_PRINT — ( [Integer Expression p. 3-338] ) ——————————|
```

Usage

The argument of ERR_PRINT( ) specifies an error message number. This must be less than zero. It is typically the value of the global **status** variable, which is affected by both SQL and 4GL errors. For SQL errors only, you can examine the global **SQLCA.SQLCODE** record. Both **SQLCA.SQLCODE** and **status** are described in "Error Handling with SQLCA" on page 2-23.

ERR_PRINT( ) is most useful when you are developing a 4GL program. The message that it returns is probably not helpful to the user of your application.

The following program segment sends any error message to the Error line:

```
LET op_status = STATUS
IF op_status < 0 THEN
    CALL ERR_PRINT(op_status)
END IF
```

(Here the value of **status** is first assigned to a variable, **op_status**, rather than calling ERR_PRINT(**status**) directly. Otherwise, the value of **status** normally would be zero, reflecting the success of the ERR_PRINT(**status**) function call.)

Like the ERR_GET( ) function (page 4-48), the ERR_PRINT( ) function cannot be invoked when the database is not ANSI-compliant, unless you first include the WHENEVER ERROR CONTINUE compiler directive to prevent the error from terminating program execution. In an ANSI-compliant database, the default action after an error has been detected is CONTINUE; see page 3-285. See also the section "Exception Handling" on page 2-23 for information about trapping run-time errors in 4GL programs.

References

ERR_GET( ), ERR_QUIT( ), ERRORLOG( ), STARTLOG( )

ERR_QUIT( )

The ERR_QUIT( ) function displays on the Error line the text of an SQL or 4GL error message, corresponding to a negative integer argument, and then terminates the program.

CALL ERR_QUIT —— (Integer Expression p. 3-338) ——|

Usage

The argument of ERR_QUIT( ) specifies an error message number. This must be less than zero. It is typically the value of the global **status** variable, which is affected by both SQL and 4GL errors. For SQL errors only, you can examine the global **SQLCA.SQLCODE** record. Both **SQLCA.SQLCODE** and **status** are described in "Error Handling with SQLCA" on page 2-23.

The ERR_QUIT( ) function is identical to the ERR_PRINT( ) function, except that ERR_QUIT( ) terminates execution once the message is printed. ERR_QUIT( ) is primarily useful when you are developing a 4GL program. The message that it returns is probably not helpful to the user of your application.

If an error occurs, the following statements display the error message on the Error line, and then terminate program execution:

```
IF STATUS < 0 THEN
    CALL ERR_QUIT(STATUS)
END IF
```

The ERR_QUIT( ) function cannot be invoked when the database is not ANSI-compliant, unless you first include the WHENEVER ERROR CONTINUE compiler directive to prevent the error from terminating program execution.

If you specify the WHENEVER ANY ERROR CONTINUE compiler directive (or equivalently, the **anyerr** command-line flag), then **status** is reset after certain additional 4GL statements, as described on page 3-283. See also the section "Exception Handling" on page 2-23 for information about trapping run-time errors in 4GL programs, and about fatal errors that cannot be trapped.

References

ERR_GET( ), ERR_PRINT( ), ERRORLOG( ), STARTLOG( )

ERRORLOG( )

The ERRORLOG( ) function copies its argument into the current error log file.

CALL ERRORLOG —— (Character Expression p. 3-343) ——————————

Usage

If you simply invoke the STARTLOG( ) function, error records that **4GL** appends to the error log after each subsequent error have this format:

```
Date: 03/06/94   Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX column.
SYSTEM error number -100
ISAM error:  duplicate value for a record with unique key.
```

You can use the ERRORLOG( ) function to supplement default error records with additional information. Entries that ERRORLOG( ) makes in the error log file automatically include the date and time when the error was recorded.

This is a typical sequence of steps for logging system error messages:

1. Use a WHENEVER ERROR CONTINUE compiler directive (page 3-281).
2. Call STARTLOG( ) to open or create an error log file (page 4-83).
3. Test the value of the global **status** variable to see if it is negative.
4. If this value is < 0, call ERR_GET( ) to retrieve the error text (page 4-51).
5. Call ERRORLOG( ) to make an entry into the error log file.

Note: *In an ANSI-compliant database, WHENEVER ERROR CONTINUE is in effect by default, unless you specify some other action for error conditions.*

You can use the ERRORLOG( ) function to identify errors in programs that you are developing and to customize error handling. Even after implementation, some errors, such as those relating to permissions and locking, are sometimes unavoidable. These can be trapped and recorded by these logging functions.

Error logging functions can be used together with other 4GL features for *instrumenting* a program, by tracking the way that the program is used. This is not only valuable for improving the program, but also for recording work habits and detecting attempts to breach security. See the *INFORMIX-4GL by Example* book for a detailed example of a program with this functionality.

The following program fragment calls STARTLOG( ) in the MAIN program block. Here the ERRORLOG( ) function has a string constant argument:

```
   CALL STARTLOG("\\usr\\catherine\\error.log")
   ...
   FUNCTION start_menu()
   CALL ERRORLOG("Entering start_menu function")
```

For a database that is not ANSI-compliant, WHENEVER ERROR CONTINUE can prevent the first SQL error from terminating program execution; but this compiler directive suppresses the automatic recording of errors by the STARTLOG( ) built-in function. If WHENEVER ERROR CONTINUE is in effect, you can make explicit calls to ERRORLOG( ), however, to maintain the error log. (The other options of WHENEVER, namely STOP, CALL, and GOTO, do not interfere with the logging of STARTLOG( ) error records.)

The following example illustrates the use of ERR_GET( ), ERRORLOG( ), and the WHENEVER ERROR CONTINUE compiler directive. It assumes that an error log file has already been created or initialized by STARTLOG( ):

```
FUNCTION add_cust()
   DEFINE errvar CHAR(80)
   WHENEVER ERROR CONTINUE
   INPUT BY NAME gr_customer.*
   INSERT INTO customer VALUES (gr_customer.*)
   IF STATUS < 0 THEN
      LET errvar = ERR_GET(STATUS)
      CALL ERRORLOG(errvar CLIPPED)
   END IF
END FUNCTION
```

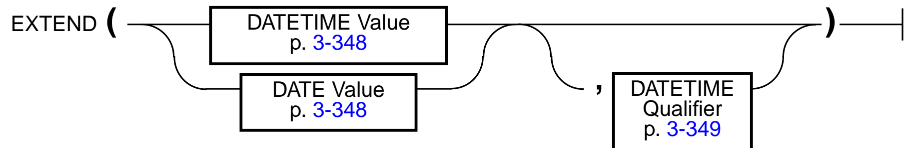
If its argument is not a character data type (for example, if it is a DECIMAL variable), invoking the ERRORLOG( ) function may produce a fatal error.

References

ERR_GET( ), ERR_PRINT( ), ERR_QUIT( ), STARTLOG( )

EXTEND( )

The EXTEND( ) operator converts an expression that returns a DATETIME or DATE value to a DATETIME value of a specified precision and scale.



Usage

The EXTEND( ) operator returns the value of its DATE or DATETIME operand, but with an adjusted precision that you can specify by a DATETIME qualifier. The operand can be a time expression of any valid precision. If it is a character string, it must consists of valid and unambiguous time unit values and separators, but with these restrictions:

- It cannot be a character string in DATE format, such as `"12/12/93"`.
- It cannot be an ambiguous numeric DATETIME value, such as `"05:06"` or `"05-06"` whose time units are ambiguous.
- It cannot be a time expression that returns an INTERVAL value.

DATETIME Qualifiers

A qualifier can specify the precision of the result (and the scale, if FRACTION is the last keyword in the qualifier). The qualifier follows a comma, and is of the form *first* TO *last*, where *first* and *last* are keywords to specify (respectively) the largest and smallest time unit in the result. Both can be the same.

If no qualifier is specified, then the following defaults are in effect, based on the explicit or default precision of the DATE or DATETIME operand:

- The default qualifier that EXTEND( ) applies to a DATETIME operand is YEAR TO FRACTION(3).
- The default qualifier for a DATE operand is YEAR TO DAY.

The section "DATETIME Qualifier" on page 3-349 describes the syntax of the YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, and FRACTION keywords in DATETIME qualifiers.

The following rules are in effect for DATETIME qualifiers that you specify as EXTEND( ) operands:

- If a *first* TO *last* qualifier is specified, the *first* keyword must specify a time unit that is more significant than or equal to the *last* keyword.
- If the *first* keyword specifies a time unit larger than any in the operand, omitted unit(s) are filled with values from the system clock-calendar. In the following fragment, the *first* keyword specifies a time unit more significant than any in **t_stamp**, so the current year would be used.

```
    DEFINE t_stamp DATETIME MONTH TO DAY
    DEFINE annual DATETIME YEAR TO MINUTE
...
    LET t_stamp = "1993-12-04 17"
    LET annual = EXTEND(t_stamp, YEAR TO MINUTE)
```

- If the *last* keyword specifies a smaller time unit than any in the operand, the new time units are filled in with constant values according to the following rules: A missing MONTH or DAY is filled in with 1, and any missing HOUR, MINUTE, SECOND, or FRACTION is filled in with 0.
- If the operand contains time units outside the precision specified by the qualifier, the unspecified time units are discarded. For example, if you specify *first* TO *last* as DAY TO HOUR, then any information about MONTH in the DATETIME operand is not used in the result.

Using EXTEND with Arithmetic Operators

If the precision of an INTERVAL value includes a time unit that is not present in a DATETIME or DATE value, you cannot combine the two values directly with the addition (+) or subtraction (-) operators. You must first use the EXTEND( ) operator to return an adjusted DATETIME value on which to perform the arithmetic operation.

For example, you cannot directly subtract the 720-minute INTERVAL value in the next example from the DATETIME value that has a precision from YEAR to DAY. You can perform this calculating by using the EXTEND( ) operator:

```
EXTEND (DATETIME (1989-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE(3) TO MINUTE
   --result:  DATETIME (1989-07-31 12:00) YEAR TO MINUTE
```

Here the EXTEND( ) operator returns a DATETIME value whose precision is expanded from YEAR TO DAY to YEAR TO MINUTE. This adjustment allows 4GL to evaluate the arithmetic expression. The result of the subtraction has the extended precision of YEAR TO MINUTE from the first operand.

In the next example, fragments of a report definition use DATE values as operands in expressions that return DATETIME values. Output from these PRINT statements would in fact be the numeric date and time (page 3-351), without the DATETIME keywords and qualifiers that are included here to show the precision of the values that the arithmetic expressions return.

```
DEFINE calendar DATE
   LET calendar = "05/18/1990"
   PRINT (calendar - INTERVAL (5-5) YEAR TO MONTH)
   --result:  DATETIME (1984-12-18) YEAR TO DAY
   PRINT (EXTEND(calendar, YEAR TO HOUR)
           - INTERVAL (4 8) DAY TO HOUR)
   --result:  DATETIME (1990-05-13 16) YEAR TO HOUR
```

You cannot directly combine a DATE with an INTERVAL value whose *last* qualifier is smaller than DAY. But as the previous example shows, you can use the EXTEND( ) operator to convert the value in a DATE column or variable to a DATETIME value that includes all the fields of the INTERVAL operand.

In the next example, the INTERVAL variable **how_old** includes fields that are not present in the DATETIME variable **t_stamp**, so the EXTEND( ) operator is required in the expression that calculates the sum of their values.

```
    DEFINE t_stamp DATETIME YEAR TO HOUR
    DEFINE age DATETIME DAY TO MINUTE
    DEFINE how_old INTERVAL DAY TO MINUTE

    LET t_stamp = "1989-12-04 17"
    LET how_old = INTERVAL (28 9:25) DAY TO MINUTE
    LET age = EXTEND(t_stamp, DAY TO MINUTE) + how_old
```

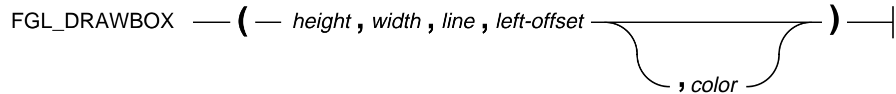
Note: *In SQL statements, you can include a similar EXTEND( ) operator of SQL, whose first argument can be the name of a DATETIME or DATE database column.*

Reference

UNITS

FGL_DRAWBOX( )

The FGL_DRAWBOX( ) function displays a rectangle of a specified size.



color is an integer expression (page 3-338) that returns a positive whole number, specifying a foreground color code.

height is an integer expression, specifying the number of screen lines occupied by the rectangle.

left-offset is an integer expression, specifying the horizontal coordinate (in characters) of the upper-left corner of the rectangle, where `1` is the first (or left-most) character in a line of the current 4GL window.

line is an integer expression, specifying the vertical coordinate of the upper-left corner, where `1` means the first (or topmost) line.

width is an integer expression, specifying the number of character positions occupied by each line of the rectangle.

Usage

The FGL_DRAWBOX( ) function draws a rectangle with the upper left corner at (*line*, *left-offset*) and the specified *height* and *width*. These dimensions must have positive integer values, in units of lines and character positions, where (`0,0`) is the upper-left corner of the current 4GL window. The optional *color* number must correspond to one of the following foreground colors:

Color Number	Foreground Color
0	WHITE
1	YELLOW
2	MAGENTA
3	RED
4	CYAN
5	GREEN
6	BLUE
7	BLACK

These are the same color options that the **upscol** utility can specify in the **syscolatt** table.

The default color is used when the color number is omitted. The color argument is optional.

Like borders, the width of the line that draws the rectangle is fixed. This fixed width cannot be specified nor modified when you invoke FGL_DRAWBOX( ). Also like borders, 4GL draws the box with the characters defined in the **termcap** or **terminfo** files. You can specify alternative characters in these files. Otherwise, 4GL uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners. If you want to assign the box a color, you must be using **termcap**, since **terminfo** does not support color. For complete information on **termcap** and **terminfo**, see Appendix F.

Rectangles drawn by FGL_DRAWBOX( ) are part of a displayed form. Each time that you execute the corresponding DISPLAY FORM or OPEN WINDOW ... WITH FORM statement, you must also redraw the rectangle.

If you invoke FGL_DRAWBOX( ) several times to create a display in which rectangles intersect, output from the most recent function call overlies any previously drawn rectangles. Screen fields and reserved lines, however, have a higher display priority than FGL_DRAWBOX( ) rectangles, regardless of the order in which the fields, lines, and rectangles are drawn.

Note: *In most applications, you should avoid drawing rectangles that intersect or overlap any field or reserved line. Reserved lines (page 3-93) may be redrawn frequently during user interaction statements, partially erasing any rectangles at the intersections where they overlap the reserved lines. To avoid this problem, position the rectangles so they do not overlap any reserved lines or screen fields.*

FGL_GETENV( )

The FGL_GETENV( ) function returns a character string, corresponding to the value of an environment variable whose name you specify as the argument.

```
FGL_GETENV ——— ( Character Expression ) ——————————————┤
                 p. 3-343
```

Usage

The argument of FGL_GETENV( ) must be a character expression that returns the name of an environment variable. For example:

- Enclose the name of the environment variable within quotes. Thus, you can identify the DBFORMAT environment variable as follows:

  ```
  fgl_getenv("DBFORMAT")
  ```

- Assign the name of the environment variable to a character variable, and then use that variable as the function argument. If you declare a CHAR or VARCHAR variable called **env_var** and then assign to it the name of an environment variable, a FGL_GETENV( ) function call could look like this:

  ```
  fgl_getenv(env_var)
  ```

If the argument is a character variable, be sure to declare it with sufficient size to store the character value returned by the FGL_GETENV( ) function. Otherwise, 4GL truncates the returned value.

If the specified environment variable is not defined, then FGL_GETENV( ) returns a NULL value. If the environment variable is defined, but does not have a value assigned to it, then FGL_GETENV( ) returns blank spaces.

You can use the FGL_GETENV( ) function anywhere within a 4GL program to examine the value of an environment variable.

The following program segment displays the value of the INFORMIXDIR environment variable. The environment variable is identified in the FGL_GETENV( ) call by enclosing the name INFORMIXDIR between quotes:

```
DEFINE path CHAR(64)
...
LET path = fgl_getenv("INFORMIXDIR")
DISPLAY "Informix installed in ", path CLIPPED
```

The next example also displays the value of the INFORMIXDIR environment variable. In this case, the environment variable is identified by the **env_var** character variable, and its contents are stored in a variable called **path**:

```
DEFINE env_var CHAR(25),
      path CHAR(64)
...
LET env_var = "INFORMIXDIR"
LET path = fgl_getenv(env_var)
DISPLAY "Informix installed in ", path CLIPPED
```

The following example examines the environment to see if the DBANSIWARN environment variable is currently set:

```
DEFINE dbansi_flag SMALLINT
...
IF (fgl_getenv("DBANSIWARN") IS NOT NULL) THEN
   LET dbansi_flag = 1
END IF
```

For descriptions of the environment variables that control features of the database engine, see the *Informix Guide to SQL: Reference*. For descriptions of environment variables that can affect the visual displays of **4GL** programs, see Appendix D.

FGL_KEYVAL( )

Function FGL_KEYVAL( ) returns the integer code of a logical or physical key.

FGL_KEYVAL —— (" Character Expression p. 3-343 ") ——|

Usage

The character expression argument of FGL_KEYVAL( ) must evaluate to one of the following names of physical or logical keys:

- Letters. (Both upper and lowercase letters are valid, but 4GL does not distinguish between them.)
- Symbols (such as !, @, and #) These must be enclosed in quotation marks.
- Any of these keywords (in uppercase or lowercase letters):

ACCEPT	HELP	NEXT *or* NEXTPAGE	RETURN *or* ENTER
DELETE	INSERT		RIGHT
DOWN	INTERRUPT	PREVIOUS *or* PREVPAGE	TAB
ESC *or* ESCAPE	LEFT		UP

F1 *through* F64

CONTROL-*char* (except A, D, H, I, J, L, M, R, or X)

Enclose the argument in quotes. If you specify a single, alphabetic character, FGL_KEYVAL( ) considers the case of the character. (This is an exception to the general rule that the **INFORMIX-4GL** language is not case-sensitive outside of quoted strings.) In all other instances, FGL_KEYVAL( ) ignores the lettercase of its argument, which you can type in either uppercase or lowercase. If the character expression argument is invalid, the function returns NULL.

***Note:** The term "key" in this section refers to a physical element of the keyboard, or to its logical effect, rather than to the SQL construct of the same name.*

Using FGL_KEYVAL( ) with FGL_LASTKEY( )

The FGL_KEYVAL( ) function can be used in form-related statements to examine a value returned by the FGL_LASTKEY( ) function, as described on page 4-62. By comparing the code returned by FGL_KEYVAL( ) with what the FGL_LASTKEY( ) function returns, you can determine whether the last key that the user pressed was a specified logical or physical key.

The FGL_KEYVAL( ) function is typically used in conditional statements and Boolean comparisons:

```
DEFINE key_var INTEGER
...
INPUT BY NAME p_customer.fname THRU p_customer.phone
   ...
   AFTER FIELD phone
      IF FGL_LASTKEY() = FGL_KEYVAL("f1") THEN
         ...
      END IF
   END INPUT
```

This example displays a message and moves the cursor to the **manu_code** field if the user presses the Up arrow key to leave the **stock_num** field:

```
CONSTRUCT query_1 ON stock.* FROM s_stock.*
   ...
   AFTER FIELD stock_num
      IF FGL_LASTKEY() = FGL_KEYVAL("up") THEN
         DISPLAY "You cannot move up from here."
         NEXT FIELD manu_code
      END IF
   ...
END CONSTRUCT
```

To determine whether the user performed some action, such as inserting a row, specify the logical name of the action (such as “INSERT”), not that of the physical key (such as “F1”). For example, the logical name of the default Accept key is ESCAPE. To test if the key most recently pressed by the user was the Accept key, you should specify FGL_KEYVAL("ACCEPT"), not FGL_KEYVAL("escape") nor FGL_KEYVAL("ESC"). Otherwise, if some key other than ESCAPE is set as the Accept key and the user presses that key, then function FGL_LASTKEY( ) does not returns a code equal to FGL_KEYVAL("ESCAPE").

The value returned by FGL_LASTKEY( ) is undefined in a MENU statement.

References

ASCII, FGL_LASTKEY( )

FGL_LASTKEY ( )

The FGL_LASTKEY( ) function returns an INTEGER code, corresponding to the logical key that the user most recently typed in a field of a screen form.

FGL_LASTKEY ( ) ——————————————————————————————┤

Usage

The FGL_LASTKEY() function returns a numeric code for the last keystroke by the user before FGL_LASTKEY( ) was called. For example, if the last key that the user entered was the lowercase `s`, the FGL_LASTKEY( ) function returns 115. Appendix A lists the numeric codes for all the ASCII characters.

The value returned by FGL_LASTKEY( ) is undefined in a MENU statement.

Note: *The term "key" in this section refers to a physical element of the keyboard of a terminal, or to its logical effect, rather than to the SQL construct of the same name.*

Using FGL_LASTKEY( ) with FGL_KEYVAL( )

You do not need to know the coding of the keys to use FGL_LASTKEY( ). The built-in FGL_KEYVAL() function, as described on page 4-60, can return a code to compare with the value returned by FGL_LASTKEY( ). The FGL_KEYVAL( ) function lets you compare the last key that the user pressed with a logical or physical key. For example, to check if the user pressed the Accept key, you would compare FGL_LASTKEY( ) with the value FGL_KEYVAL("**accept**").

For example, the following CONSTRUCT statement checks the value of the last key that the user entered in each field. If the user last pressed the RETURN key, then the program displays a message in the Error line:

```
CONSTRUCT query_1 ON stock.* FROM s_stock.*
   BEFORE CONSTRUCT
      DISPLAY "Use the TAB key to move ",
            "between the fields." AT 1,1
   AFTER FIELD stock_num, manu_code, description,
         unit_price, unit, unit_descr
      IF FGL_LASTKEY( ) = FGL_KEYVAL("return") THEN
         ERROR "Use the TAB key to move the cursor ",
               "between the fields."
      END IF
END CONSTRUCT
```

Here (as in ON KEY clauses), RETURN is a synonym for ENTER.

The following example demonstrates using the FGL_LASTKEY( ) function after a PROMPT statement that expects the user to respond to the prompt with a single keystroke. The FGL_LASTKEY( ) function returns the code of the key the user pressed to the program. The FGL_LASTKEY( ) function compares the code with the code for the RETURN key. If an exact match occurs, 4GL calls the **continue( )** function. If a match does not occur, because the user pressed a key other than RETURN, then 4GL calls the **quit( )** function:

```
DEFINE      value CHAR,
      key INTEGER

PROMPT "Press the RETURN key to continue. ",
      "Press any other key to quit." FOR CHAR value
LET key = FGL_LASTKEY()
IF key = FGL_LASTKEY("return") THEN
   CALL continue()
ELSE
   CALL quit()
END IF
```

AUTONEXT Fields

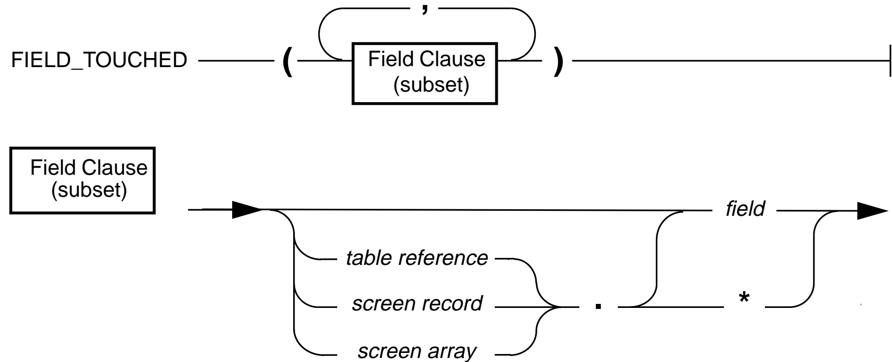
If FGL_LASTKEY( ) is invoked after the user enters a value in a field with the AUTONEXT attribute (page 5-30), 4GL returns the code of the last key that the user entered, regardless of any processing done in the AFTER FIELD or BEFORE FIELD clause.

References

ASCII, FGL_KEYVAL( )

FIELD_TOUCHED( )

The FIELD_TOUCHED( ) operator tests whether the user has made any change in a specified field (or list of fields) of the current 4GL form. (This operator can only appear within CONSTRUCT, INPUT, or INPUT ARRAY statements.)



field	is a field identifier (from the ATTRIBUTES section of the form specification file).
screen array	is the 4GL identifier that you declared for a screen array in the INSTRUCTIONS section of the form specification file.
screen record	is the 4GL identifier that you declared for a screen record.
table reference	is a table name, alias, or synonym (or FORMONLY keyword).

Usage

The FIELD_TOUCHED( ) operator returns a Boolean value TRUE (meaning that the user made a change to the field) when a DISPLAY statement displays data in any of the specified fields, or the user presses any of the following keys:

- A printable character (including the Spacebar)
- CONTROL-X (character delete)
- CONTROL-D (clear to end of field)

After any of these keystrokes, the FIELD_TOUCHED( ) operator returns TRUE, regardless of whether the keystroke has changed the value in the field.

Otherwise, the FIELD_TOUCHED( ) operator returns FALSE if none of the specified fields are edited. Moving through a field (by pressing RETURN, TAB, or the Arrow keys) does *not* mark a field as "touched."

The FIELD_TOUCHED( ) operator is valid only in CONSTRUCT, INPUT, and INPUT ARRAY statements. When you use it in an INPUT ARRAY statement, 4GL assumes that you are referring to the current screen record. You cannot specify a subscript to access a record in different row of the screen array.

This operator does not register the effect of 4GL statements that appear in a BEFORE CONSTRUCT or BEFORE INPUT clause. You can assign values to fields in these clauses without marking the fields as touched.

In the following program fragment, an IF statement tests whether the user has entered a value into any form field. If no field has been touched, the program prompts the user to indicate whether to retrieve all customer records. If the user types N or n, the CONTINUE CONSTRUCT statement is executed, and the cursor is positioned in the form, giving the user another opportunity to enter selection criteria. If the user types any other key, the program terminates the IF statement and reaches the END CONSTRUCT keywords.

```
CONSTRUCT BY NAME query1 ON customer.*
...
   AFTER CONSTRUCT
      IF NOT FIELD_TOUCHED(customer.*) THEN
         PROMPT "Do you really want to see ",
            "all customer rows? (y/n)"
            FOR CHAR answer
         IF answer MATCHES "[Nn]" THEN
            CONTINUE CONSTRUCT
         END IF
      END IF
END CONSTRUCT
```

Note: *This is not as dependable as testing whether* query1 = " 1=1" *after the END CONSTRUCT keywords, because the user may have left all the fields blank after first entering and then deleting query criteria in some field. In that case, the resulting Boolean expression (*" 1=1"*) can retrieve all rows, but FIELD_TOUCHED( ) returns TRUE, and the PROMPT statement is not executed. See* *"Searching for All Rows" on page 3-50* *for additional information.*

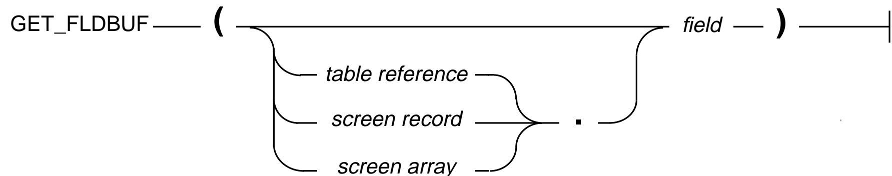
References

Boolean Operators, FGL_KEYVAL( ), FGL_LASTKEY( ), GET_FLDBUF( ), INFIELD( )

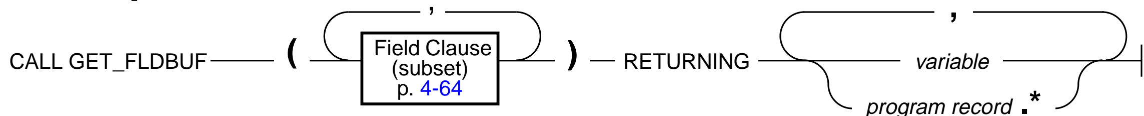
GET_FLDBUF( )

The GET_FLDBUF( ) operator returns the character values of the contents of one or more fields in the currently active screen form. (This operator can only appear within the CONSTRUCT, INPUT, or INPUT ARRAY statements of 4GL.)

Case I:
(single field)



Case II:
(multiple fields)



field	is the name of a field in the current screen form.
program record	is the name of a program record, whose members are character variables to store values from the specified fields.
screen-array	is the name of a screen array that was defined in the INSTRUCTIONS section of the form specification file.
screen-record	is the name of a screen record that is explicitly or implicitly defined in the form specification file.
table reference	is the unqualified name, alias, or synonym of a database table or view, or else the keyword FORMONLY.
variable	is a name within a list of one or more character variables, separated by commas. Variables must correspond in number and position with the list of fields in the field clause.

Usage

GET_FLDBUF( ) operates on a list of one or more fields. For example, this LET statement assigns the value in the **lname** field to the **lbuff** variable:

```
LET lbuff = GET_FLDBUF(lname)
```

To specify a list of several field names as operands of GET_FLDBUF( ), you must use the CALL statement with the RETURNING clause. Insert commas to separate successive field names and successive variables:

```
CALL GET_FLDBUF(c_num, company, lname)
   RETURNING p_cnum, p_company, p_lname
```

The following statement returns a set of character values corresponding to the contents of the **s_customer** screen record and assigns these values to the **p_customer** program record:

```
CALL GET_FLDBUF(s_customer.*) RETURNING p_customer.*
```

(The first asterisk (*) specifies all the fields in the **s_customer** screen-record; the second specifies all the members of the **s_customer** program record.)

You can use the GET_FLDBUF( ) operator to assist a user when entering a value in a field. For example, if you have an input field for last names, you can include an ON KEY clause that lets a user enter the first few characters of the desired last name. If the user calls the ON KEY clause, 4GL displays a list of last names that begin with the characters entered. The user can then choose a last name from the list. The following program fragment demonstrates this use of the GET_FLDBUF( ) operator:

```
DEFINE lname, myquery, partial_name CHAR(20),
       tw ARRAY[10] OF CHAR(20),
       a INTEGER
...
INPUT BY NAME lname
   ON KEY (CONTROL-P)
      LET partial_name = GET_FLDBUF(lname)
      LET myquery = "SELECT lname FROM teltab ",
         "WHERE lname MATCHES \"", partial_name CLIPPED, "*\""
      OPEN WINDOW w1 AT 5,5 WITH FORM "tel_form"
         ATTRIBUTE (BORDER)
      DISPLAY partial_name AT 1,1
      PREPARE mysubquery FROM myquery
      DECLARE q1 CURSOR FOR mysubquery
      LET a = 0
      FOREACH q1 INTO lname
         LET a = a + 1
         ...
      END FOREACH
      DISPLAY a TO ncount
      IF (a = 0) THEN
         PROMPT "Nothing beginning with these letters"
```

```
FOR CHAR partial_name
      ELSE
         IF (a > 10) THEN  LET a = 10
         END IF
         CALL SET_COUNT(a)
         DISPLAY ARRAY tw TO srec.*
      END IF
   ...
END INPUT
```

If you assign the character string returned by the GET_FLDBUF( ) operator to a variable that is not defined as a character data type, 4GL tries to convert the string to the appropriate data type. Conversion is not possible in these cases:

- The field contains special characters (for example, date or currency characters) that 4GL cannot convert.
- The GET_FLDBUF( ) operator is called from a CONSTRUCT statement and the field contains comparison or range operators that 4GL cannot convert.

GET_FLDBUF( ) is valid only in CONSTRUCT, INPUT, and INPUT ARRAY statements. When it encounters this operator in an INPUT ARRAY statement, 4GL assumes that you are referring to the current row. You cannot use a subscript within brackets to reference a different row of the screen array.

The following example uses the GET_FLDBUF( ) and FIELD_TOUCHED( ) operators in an AFTER FIELD clause in a CONSTRUCT statement. The FIELD_TOUCHED( ) operator checks whether the user has entered a value into the **zipcode** field. If FIELD_TOUCHED( ) returns TRUE, then GET_FLDBUF( ) retrieves the value entered into the field and assigns it to the **p_zip** program variable. If the first character in the **p_zip** variable is not a 9, the program displays an error, clears the field, and returns the cursor to the field.

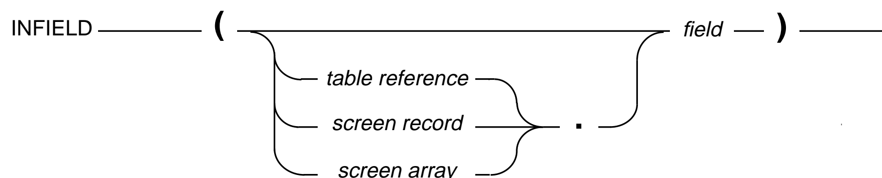
```
CONSTRUCT BY NAME query1 ON customer.*
   ...
   AFTER FIELD city
      IF FIELD_TOUCHED(zipcode) THEN LET p_zip = GET_FLDBUF(zipcode)
         IF p_zip[1,1] <> "9" THEN
            ERROR "You can only search in section 9."
            CLEAR zipcode
            NEXT FIELD zipcode
         END IF
      END IF
```

References

FIELD_TOUCHED( ), INFIELD( )

INFIELD( )

The INFIELD( ) operator in CONSTRUCT, INPUT or INPUT ARRAY statements tests whether its operand is the identifier of the current screen field.



field	is the name of a field in the current screen form.
screen-array	is the name of a screen array that you defined in the INSTRUCTIONS section of the form specification file.
screen-record	is the name of a screen record that is explicitly or implicitly defined in the form specification file.
table reference	is the unqualified name, alias, or synonym of a database table or view, or else the keyword FORMONLY.

Usage

INFIELD( ) is a Boolean operator that returns the value TRUE if *field* is the name of the current screen field. Otherwise, INFIELD( ) returns the value FALSE. (For information on assigning a name to a display field of a screen form, see "ATTRIBUTES Section" on page 5-20.)

***Note:** You must specify a field name, rather than a field tag, as the operand.*

You can use INFIELD( ) during a CONSTRUCT, INPUT, or INPUT ARRAY statement to take field-dependent actions. The INFIELD( ) operator is typically part of an ON KEY clause, often in conjunction with the built-in function SHOWHELP( ) to display Help messages to the user.

The next code fragment is from a program that uses INFIELD( ) to determine whether to display a popup window:

```
ON KEY (CONTROL-F, F5)
   IF INFIELD(customer_num) THEN
      CALL cust_popup()
```

When a user presses either of two keys during the INPUT, the **cust_popup( )** function is invoked if the screen cursor is in the **customer_num** field.

In the following fragment, **call_flag** and **res_flag** are the names of fields:

```
ON KEY (F2, CONTROL-E)
   IF INFIELD(call_flag) OR INFIELD(res_flag) THEN
      IF INFIELD (call_flag) THEN
         LET fld_flag = "C"
      ELSE        --* user pressed F2 (CTRL-E) from res_flag
         LET fld_flag = "R"
      END IF
   ...
   END IF
```

Subsequent code could use these field names to determine which column of a row to edit.

In the following example, the INPUT statement uses the INFIELD( ) operator with the SHOWHELP( ) function to display field-dependent Help messages.

```
   INPUT gr_equip.* FROM sr_equip.*
       ON KEY(CONTROL-B)
           CASE
               WHEN INFIELD(part_num)
                   CALL SHOWHELP(301)
               WHEN INFIELD(part_name)
                   CALL SHOWHELP(302)
               WHEN INFIELD(supplier)
                   CALL SHOWHELP(303)
               ...
           END CASE
   END INPUT
```

References

SCR_LINE( ), SHOWHELP( ), FIELD_TOUCHED( ), GET_FLDBUF( )

LENGTH( )

The LENGTH( ) function returns the number of bytes in its string argument after deleting all trailing spaces.

```
LENGTH ——— ( — Character Expression p. 3-343 — ) ———————————|
```

Usage

The LENGTH( ) function returns an integer value, based on the length (in bytes) of its character-expression argument.

Statements in the next example center a report title on an 80-column page:

```
   LET title = "Invoice for ", fname CLIPPED,
                     " ", lname CLIPPED
   LET offset = (80 - length(title))/2
   PRINT COLUMN offset, title
```

The following are among the possible uses for the LENGTH( ) function:

- You can check whether a user has entered a database name and, if not, set a default name.
- Check whether the user has supplied the name of a file to receive the output from a report and, if not, set a default output.
- Use LENGTH(*string*) as the MAX( ) value in a FOR loop, and then check each character in *string* for a specific character. For example, you can check for "." to determine whether a table name has a prefix.

LENGTH( ) is also useful as a check on user input. In the following example, an IF statement is used to determine whether the user has responded to a displayed message:

```
IF LENGTH (ans1) = 0 THEN
   PROMPT "Press RETURN to continue: " FOR input_val
ELSE ...
```

Using LENGTH( ) in SQL Expressions

Unlike some other built-in functions of 4GL, you can use LENGTH( ) in SQL statements, as well as in other 4GL statements. LENGTH( ) can also be called from a C function. (That is, the database engine supports a function of the same name and of similar functionality.)

In a SELECT or UPDATE statement, the argument of LENGTH( ) is the identifier of a character column. In this context, LENGTH( ) returns the number of bytes in the CLIPPED data value (for CHAR or VARCHAR columns) or the full number of bytes (for TEXT and BYTE data types).

The LENGTH( ) function can also take the name of a database column as its argument, but only within an SQL statement.

SE

With the INFORMIX-SE engine, the LENGTH( ) function cannot reference columns that correspond to VARCHAR, TEXT, nor BYTE data types.

References

CLIPPED, USING

LINENO

The LINENO operator returns the number of the line within the page that is currently printing. (This operator can appear only in the FORMAT section of a REPORT program block.)

```
LINENO ──────────────────────────────────────────────────────┤
```

Usage

This returns the value of the *line number* of the report line that is currently printing. 4GL computes the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN.

For example, the following program fragment examines the value of LINENO. If this value is less than 9, a PRINT statement formats and displays it, beginning in the tenth character position after the left margin.

```
IF (LINENO > 9) THEN
   PRINT COLUMN 10, LINENO USING "Line <<<"
END IF
```

You can specify LINENO in the PAGE HEADER, PAGE TRAILER, or other report control blocks, to find the print position on the current page of a report.

INFORMIX-4GL cannot evaluate the LINENO operator outside the FORMAT section of a REPORT program block. The value that LINENO returns must be assigned to a variable that is not local to the report, if you need to reference this value within some other program block of your 4GL application.

Reference

PAGENO

MDY( )

The MDY( ) operator returns a value of the DATE data type from three integer operands that represent the *month*, the *day* of the month, and the *year*.

MDY (— Integer Expression p. 3-338 — , — Integer Expression p. 3-338 — , — Integer Expression p. 3-338 —)

Usage

The MDY( ) operator converts to a single DATE format a list of exactly three valid integer expressions. The three expressions correspond with the *month*, *day*, and *year* elements of a calendar date:

- The first expression must return an integer, representing the number of the month (1-12).
- The second expression must return an integer, representing the number of the day of the month (1-28, 29, 30, or 31, depending on the month).
- The third expression must return a four-digit integer, representing the year.

An error results if you specify values outside the range of days and months in the calendar, or if the number of operands is not three.

You must enclose the three integer expression operands within parentheses, separated by commas, just as you would if MDY( ) were a function.

The value of the third expression cannot be the abbreviation for the year. For example, `91` specifies a year in the first century, more than 1,900 years ago.

The following program uses MDY( ) to return a DATE value, which is then assigned to a variable and displayed on the screen:

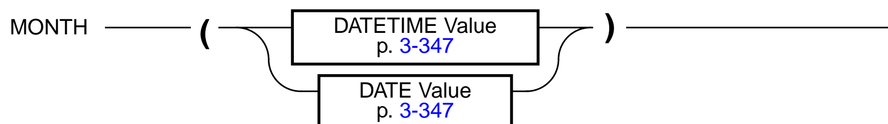
```
MAIN
DEFINE a_date DATE
LET a_date = MDY(12/2,3+2,1988)
DISPLAY a_date
END MAIN
```

Reference

DATE( )

MONTH( )

The MONTH( ) operator returns a positive whole number between 1 and 12, corresponding to the *month* portion of a DATE or DATETIME operand.



Usage

The MONTH( ) operator extracts an integer value for the month in a DATE or DATETIME value. You cannot specify an INTERVAL operand.

The following program extracts the month time unit from a DATETIME literal expression. It evaluates MONTH(**date_var**) as an operand of a Boolean expression to test whether the month is earlier in the year than March.

```
MAIN

DEFINE date_var DATETIME YEAR TO SECOND
DEFINE current_month CHAR(10)
DEFINE month_var INT

LET current_month = CURRENT MONTH TO MONTH
LET date_var = DATETIME(89-01-12 18:47:32) YEAR TO SECOND
LET month_var = MONTH(date_var)
DISPLAY "The current month is: ", current_month
DISPLAY "The month of interest is month number : ",
     month_var USING "##"
IF MONTH(date_var) < 3
    THEN DISPLAY "Month of interest is Feb. or Jan.”
END IF

END MAIN
```

References

DATE( ), DAY( ), WEEKDAY( ), YEAR( )

NUM_ARGS( )

The NUM_ARGS( ) function returns the number of command-line arguments with which the current 4GL program is invoked.

```
NUM_ARGS ( ) ------------------------------------------------|
```

Usage

The NUM_ARGS( ) function returns an integer, corresponding to the number of command-line arguments that followed the name of your 4GL program when the user invoked it.

You can use the ARG_VAL( ) built-in function to retrieve individual arguments. By using NUM_ARGS( ) in conjunction with the ARG_VAL( ) function (page 4-16), the program can pass command-line arguments to the MAIN statement, or to whatever program block invokes the NUM_ARGS( ) and ARG_VAL( ) functions.

In the following examples, each of the command lines includes three arguments:

```
myprog.4gi kim sue joe          (C Compiler version)
fglgo myprog kim sue joe        (RDS version)
```

After either of these command lines, NUM_ARGS( ) sets 3 as the upper limit of variable **i** in the FOR loop of the program fragment that follows.

```
DEFINE pa_args ARRAY[8] OF CHAR(10),
       i     SMALLINT
FOR i = 1 TO NUM_ARGS()
       LET pa_args[i] = ARG_VAL(i)
END FOR
```

Reference

ARG_VAL( )

PAGENO

The PAGENO operator returns a positive whole number, corresponding to the number of the page of report output that 4GL is currently printing. (PAGENO is valid only in the FORMAT section of a REPORT program block.)

PAGENO ————————————————————————————┤

Usage

This returns a positive integer, whose value is the number of the page that includes the current print position in the currently executing report.

For example, the following program fragment conditionally prints the value returned by PAGENO, using the USING operator to format it, if this value is less than 10,000.

```
IF (PAGENO < 10000) THEN
   PRINT COLUMN 28, PAGENO USING "page <<<<"
END IF
```

You can include the PAGENO operator in PAGE HEADER or PAGE TRAILER control blocks or in other control blocks of a report definition, to identify the page numbers of output from a report.

4GL cannot evaluate the PAGENO operator outside the FORMAT section of a REPORT program block. If some other program block of your 4GL application needs to reference the value that PAGENO returns, the report must assign that value to a variable whose scope of reference is not local to the report.

Reference

LINENO

SCR_LINE( )

The SCR_LINE( ) function returns a positive integer, corresponding to the number of the current screen record in its screen array during a DISPLAY ARRAY or INPUT ARRAY statement.

```
SCR_LINE ( ) ──────────────────────────────────────────┤
```

Usage

The current screen record is the line of a screen array that contains the screen cursor at the beginning of a BEFORE ROW or AFTER ROW clause.

The first record of the program array and of the screen array are both numbered 1. The built-in 4GL functions SCR_LINE( ) and ARR_CURR( ) can return different values if the program array is larger than the screen array.

The following program fragment tests what the user enters, and rejects it if the **state** field value indicates that the customer is not from California.

```
DEFINE pa_clients ARRAY[90] OF RECORD
    fname   CHAR(15),
    lname   CHAR(15),
    state   CHAR(2)
    END RECORD,
    curr_pa, curr_sc SMALLINT

INPUT ARRAY pa_clients FROM sa_clients.*
    AFTER FIELD state
        LET curr_pa = ARR_CURR()
        LET curr_sc = SCR_LINE()
        IF UPSHIFT(pa_clients[curr_pa].state) != "CA" THEN
            ERROR "Policy for California clients only"
            INITIALIZE pa_clients[curr_pa].* TO NULL
            CLEAR scr_array[curr_sc].*
            NEXT FIELD fname
        END IF
END INPUT
```

The following example makes use of SCR_LINE( ) and of the related ARR_CURR( ) built-in function to assign values to variables within the BEFORE ROW clause of an INPUT ARRAY statement. Because these functions

are invoked in the BEFORE ROW control block, the respective **curr_pa** and **curr_sa** variables are evaluated each time that the cursor moves to a new line, and are available within other clauses of the INPUT ARRAY statement.

```
INPUT ARRAY ga_items FROM sa_items.* HELP 62
   BEFORE ROW
      LET curr_pa = ARR_CURR()
      LET curr_sa = SCR_LINE()
```

It is then possible, for example in a later statement within INPUT ARRAY, to have a statement such as the following, which fills in the **description** and **unit_price** fields on the screen:

```
DISPLAY
   ga_items[curr_pa].description, ga_items[curr_pa].unit_price
TO
   sa_items[curr_sa].description, sa_items[curr_sa].unit_price
```

References

ARR_COUNT( ), ARR_CURR( )

SET_COUNT( )

The SET_COUNT( ) function specifies the number of records that contain data in a program array.

CALL SET_COUNT —— (— Integer Expression p. 3-338 —) ——————|

Usage

Before you use an INPUT ARRAY WITHOUT DEFAULTS or a DISPLAY ARRAY statement, you must call the SET_COUNT( ) function with an integer argument to specify the total number of records in the program array. In typical applications, these records contain retrieved values from the database.

The SET_COUNT( ) built-in function sets an initial value from which the ARR_COUNT( ) function determines the total number of members in an array. If you do not explicitly call ARR_COUNT( ), a default value of zero is assigned.

In the following program fragment, the variable **n_rows** is an array index that received its value in an earlier FOREACH loop. The index was initialized with a value of 1, so the expression (**n_rows** -1) represents the number of rows that were fetched from a database table in the FOREACH loop. The expression SET_COUNT (**n_rows** - 1) tells INPUT ARRAY WITHOUT DEFAULTS how many program records containing row values from the database are in the program array, so it can determine how to control the screen array.

```
CALL SET_COUNT(n_rows - 1)
INPUT ARRAY pa_items WITHOUT DEFAULTS
    FROM sa_items.*
```

If no INPUT ARRAY statement has been executed, and you do not call the SET_COUNT ( ) function, then the DISPLAY ARRAY or INPUT ARRAY WITHOUT DEFAULTS statement displays no records.

References

ARR_COUNT( ), ARR_CURR( )

SHOWHELP( )

The SHOWHELP( ) function displays a Help message, corresponding to its specified SMALLINT argument, from the current Help file.

CALL SHOWHELP —— (—[Integer Expression p. 3-338]—) ——————————|

Usage

The argument of SHOWHELP( ) identifies the number (between 1 and 32,767) of a message in the Help file that was specified in the OPTIONS statement.

When it is called, SHOWHELP( ) opens the Help window (as described on page 2-22), and displays the first (or only) page of the Help message text below a ring menu of Help options. This menu is called the Help menu.

If the Help message is too long to fit on one page, the **Screen** option of the Help menu can display the next page of the message. The **Resume** option closes the Help window, and returns focus to the 4GL screen. (See also the description of the **mkmessage** utility in Appendix B.)

In interactive statements like CONSTRUCT, INPUT, INPUT ARRAY, PROMPT, or the COMMAND clause of a MENU statement, the effect of SHOWHELP( ) resembles that of the Help key. The Help key, however, displays only the message specified in the current HELP clause. The following example uses INFIELD( ) with SHOWHELP( ) to display field-dependent Help messages:

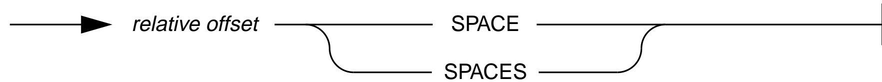
```
INPUT ARRAY gr_equip.* FROM sa_equip.*
    ON KEY(CONTROL-B)
        CASE
            WHEN INFIELD(part_num)
                CALL SHOWHELP(301)
            WHEN INFIELD(part_name)
                CALL SHOWHELP(302)
            WHEN INFIELD(supplier)
                CALL SHOWHELP(303)
            ...
        END CASE
END INPUT
```

Reference

INFIELD( )

SPACE

The SPACE operator returns a string of a specified length, containing only blank (ASCII 32) characters. The keyword SPACES is a synonym for SPACE.



relative offset is an integer expression (page 3-338) that returns a positive number. This specifies the number of blank characters.

Usage

This returns a blank string of length *relative offset*, identical to a quoted string containing the same number of blanks.

In a PRINT statement in the FORMAT section of a report definition, the SPACE operator advances the character position by *relative offset* characters.

The following statements from a fragment of a report definition use the SPACE operator to accomplish several tasks:

- To separate variables within two PRINT statements.
- To concatenate six (6) blank spaces to the string "`=ZIP`".
- To print the resulting string after the value of the variable **zipcode**.

```
FORMAT
  ON EVERY ROW
     LET mystring = (6 SPACES), "=ZIP"
     PRINT fname, 2 SPACES, lname
     PRINT company
     PRINT address1
     PRINT city, ", " , state, 2 SPACES, zipcode, mystring
```

In a DISPLAY statement, the SPACE operator inserts *relative offset* blank characters into the output.

Outside PRINT statements, the *relative offset* operand and the SPACE (or SPACES) keyword must appear within parentheses, as in the LET statement of the previous example.

References

LINENO, PAGENO

SQLEXIT( )

The SQLEXIT( ) function terminates the connection of the **4GL** application to the database server, and returns a status code of zero. (No compilation error occurs if your application ignores this returned value.)

```
CALL SQLEXIT ( ) ──────────────────────────────────────┤
```

Usage

SQLEXIT( ) rolls back any open transaction, closes the current database, and terminates the database connection. Its effect resembles that of the DISCONNECT ALL statement of SQL. One difference is that DISCONNECT ALL fails if there are any current transactions. If the current database is remote, or if it uses transaction logging, first issue the CLOSE DATABASE statement before calling SQLEXIT( ). To verify that the rollback or database closure succeeded, examine the SQLCA record (page 2-23).

The following fragment of a MENU statement calls the SQLEXIT( ) function:

```
COMMAND  "Disconnect"
   CLOSE DATABASE
   CALL SQLEXIT() RETURNING l_oeuf
   HIDE OPTION ALL
   SHOW OPTION "Connect"
```

If your **4GL** code enables the user to invoke SQLEXIT( ), as in this example, and if it also traps error **-854**, then you can create groups of **4GL** applications that can emulate running concurrently in different **4GL** screens while alternately using the same connection to one or more databases.

Another use of SQLEXIT( ) is to disconnect from a local engine, so that you can reset environment variables like DBDATE. After you change the environment variables, use the DATABASE statement to reconnect to the database server.

You can also use SQLEXIT( ) to reduce database overhead in **4GL** programs that refer to a database only briefly and after long intervals, or that access a database only during initialization.

STARTLOG( )

The STARTLOG( ) function opens an error log file.

```
CALL STARTLOG ——(——— "filename" ———)————————————————|
                  \___ variable ___/
```

filename is a quoted string to specify a filename (and optional pathname and file extension) of the error log file.

variable is a variable of type CHAR or VARCHAR containing a filename (and optional pathname and file extension) of the error log file.

Usage

The following is a typical sequence to implement error logging:

1. Call STARTLOG( ) in the MAIN program block to open or create an error log file.
2. Use a LET statement with ERR_GET(**status**) to retrieve the error text (page page 4-48) and to assign this value to a program variable.
3. Use ERRORLOG( ) to make an entry into the error log file (page page 4-51).

The last two steps are not needed, if you are satisfied with the error records that are automatically produced after STARTLOG( ) has been invoked. After you call the STARTLOG( ) function, a record of every subsequent error that occurs during the execution of your program is written to the error log file, provided that the WHENEVER ERROR CONTINUE statement (page 3-281) is not in effect.

The error record consists of the date, time, source-module name and line number, error number, and error message. You can also write your own messages in the error log file by using the ERRORLOG( ) function (page 4-51). If you invoke the STARTLOG( ) function, error records that **4GL** appends to the error log file after each subsequent error have the following format:

```
Date: 03/06/94   Time: 12:20:20
Program error at "stock_one.4gl", line number 89.
SQL statement error number -239.
Could not insert new row - duplicate value in a UNIQUE INDEX column.
SYSTEM error number -100
ISAM error:  duplicate value for a record with unique key.
```

With other 4GL features, the STARTLOG( ), ERR_GET( ), and ERRORLOG( ) functions can be used for *instrumenting* a program, to track how the program is used. This is not only valuable for improving the program, but also for recording work habits and detecting attempts to breach security. Example 25 in *INFORMIX-4GL by Example* contains an example of this type of functionality.

The following program fragment invokes the STARTLOG( ) built-in function, specifying the name of the error log file in a quoted string that includes a pathname and a file extension. The function definition includes a call to the built-in ERRORLOG( ) function, which adds a message to the error log file.

```
CALL STARTLOG("/usr/arik/error.log")
...
FUNCTION start_menu()
CALL ERRORLOG("Entering start_menu function")
...
END FUNCTION
```

In a database that is ANSI-compliant, WHENEVER ERROR CONTINUE is the default error-handling action when a run-time error condition is detected.

If the database is not ANSI-compliant, WHENEVER ERROR STOP is the default error-handling action. WHENEVER ERROR CONTINUE can prevent the first SQL error from terminating program execution. Another effect of this compiler directive, however, is to suppress automatic recording of errors by STARTLOG( ). If WHENEVER ERROR CONTINUE is in effect, you can use explicit calls to the ERRORLOG( ) function to maintain the error log. (The other options of WHENEVER, namely STOP, CALL, and GOTO, do not interfere with automatic logging of STARTLOG( ) error records.)

Specifying the Error Log File

If the argument of STARTLOG( ) is not the name of an existing file, then STARTLOG( ) creates one. If the file already exists, STARTLOG( ) opens it, and positions the file pointer so that subsequent error messages can be appended to this file. For portable programs, the filename should be a variable, rather than a literal string.

References

ERR_GET( ), ERRORLOG( )

TIME

The TIME operator reads the system clock and returns a character string representing the current time-of-day.

```
TIME ──────────────────────────────────────────┤
```

Usage

TIME returns a character string, representing the current time in the format *hh:mi:ss*, based on a 24-hour clock. (Here *hh* represents the *hour*, *mi* the *minute*, and *ss* the *second* as 2-digit strings, with colon (:) symbols as separators.)

In the following example, the value returned by TIME is assigned to the **p_time** variable and displayed:

```
DEFINE p_time char(15)
LET p_time = TIME

DISPLAY "The time is ", p_time
```

References

CURRENT, DATE, TODAY

TODAY

The TODAY operator reads the system clock and returns a DATE value that represents the current calendar date.

TODAY

Usage

TODAY is typically used in LET and PRINT statements to record the current date in situations where the time of day (which CURRENT or TIME supplies) is not necessary. Like the DATE, TIME, and CURRENT operators, the effect of TODAY is sensitive to the time of execution and the accuracy of the system clock-calendar.

NLS

4GL can display language-specific month name and day name abbreviations. This requires the installation of message files in a subdirectory of **$INFORMIXDIR/msg**, and subsequent reference to that subdirectory by way of the environment variable DBLANG. For example, the *weekday* portion of the date string in a Spanish locale translates the day Saturday into the day name abbreviation Sab, which stands for Sabado (the Spanish word for Saturday). For more information on NLS, see Appendix E.

The following example uses TODAY in a REPORT definition:

```
SKIP 1 LINE
PRINT COLUMN 15, "FROM: ", begin_date USING "mm/dd/yy",
    COLUMN 35, "TO: ", end_date USING "mm/dd/yy"
PRINT COLUMN 15, "Report run date:  ",
    TODAY USING "mmm dd, yyyy"
SKIP 2 LINES
PRINT COLUMN 2, "ORDER DATE", COLUMN 15, "COMPANY",
    COLUMN 35, "NAME", COLUMN 57, "NUMBER",
    COLUMN 65, "AMOUNT"
```

TODAY is sometimes useful in setting defaults and initial values in form fields. The next code fragment initializes a field with the current date if the field is empty. This takes place before the user enters data into the field:

```
INPUT gr_payord.paid_date FROM a_date
   BEFORE FIELD a_date
      IF gr_payord.paid_date IS NULL THEN
         LET gr_payord.paid_date = TODAY
      END IF
```

References

CURRENT, DATE, DATE( ), TIME

UNITS

The UNITS operator returns an INTERVAL value, based on a single time unit.

```
Integer Expression ─── UNITS ─── keyword ───────────────┤
p. 3-338
```

keyword is one of the following keywords to specify a time unit:

YEAR	MONTH
DAY	HOUR
MINUTE	SECOND
	FRACTION (*n*)

Here *n* is an integer, greater than zero but less than 6, to specify the scale of the second.

Usage

The UNITS operator returns an INTERVAL value for a single unit of time, such as DAY TO DAY, YEAR TO YEAR, or HOUR TO HOUR. If you substitute a number expression for the integer operand, any fractional part of the returned value is discarded before the UNITS operator is applied.

The UNITS operator has a higher precedence than any arithmetic or Boolean operator. You may need parentheses around its operand, if this is an expression that includes other operators. Any left-hand arithmetic operand that includes the UNITS operator must be enclosed within parentheses.

In the following example, the user has specified a starting time for a meeting (DATETIME value) as well as a value for the length of the meeting, which the program has already converted to minutes (SMALLINT). Here the program computes when the meeting will end (DATETIME value). UNITS in this case allows you to add the SMALLINT value to the DATETIME value and get a new DATETIME value.

```
LET end_time = (start_time + meeting_length) UNITS MINUTE
```

Because the difference between two DATE values is an integer count of days (page 4-18) rather than an INTERVAL data type, you may wish to use the UNITS operator to convert such differences explicitly to INTERVAL values:

```
LET lateness = (date_due - TODAY) UNITS DAYS
```

References

Arithmetic Operators

UPSHIFT( )

The UPSHIFT( ) function takes a character-string argument and returns a string in which any lowercase letters are converted to uppercase letters.

UPSHIFT —— (—[Character Expression p. 3-343]—) ——|

Usage

The UPSHIFT( ) function is most often used to regularize data, for example, to prevent the state abbreviation “VA,” “Va,” or “va” from resulting in different values, if these were logically equivalent in the context of your application.

You can use the UPSHIFT( ) function in an expression where a character string is valid, in DISPLAY and PRINT statements, and in assignment statements. (See also the UPSHIFT field attribute on page 5-54.)

Non-alphabetic or uppercase characters are not altered by UPSHIFT( ). The maximum data length of the returned character string value is 511 bytes.

NLS

When NLS is active, the results of conversion between uppercase and lowercase are appropriate to the national language in use, as defined by the LC_CTYPE environment variable.

The following example demonstrates a function that was written to merge two privilege strings. Its output preserves letters in preference to hyphens (privileges over lack of privilege) and uppercase letters in preference to lowercase (privileges WITH GRANT OPTION over those without).

```
FUNCTION merge_auth(oldauth, newauth)
   DEFINE oldauth, newauth LIKE systabauth.tabauth, k SMALLINT
   FOR k = 1 TO LENGTH(oldauth)
      IF (oldauth[k] = "-")   -- no privilege in this position
         OR (UPSHIFT(oldauth[k]) = newauth[k])
                              -- new is "with grant option"
      THEN LET oldauth[k] = newauth[k])
      END IF
   END FOR
   RETURN oldauth
END FUNCTION
```

In the next example, the CHAR variables **u_str** and **str** are equivalent, except that **u_str** substitutes uppercase letters for any lowercase letters in **str**.

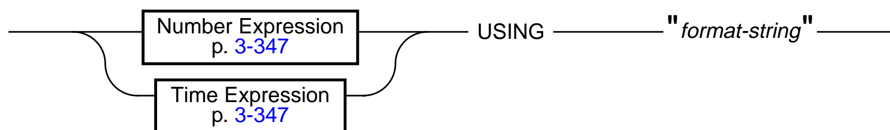
```
LET u_str = UPSHIFT(str)
```

Reference

DOWNSHIFT( )

USING

The USING operator specifies a character-string format for a number, MONEY, or DATE operand, and returns the formatted value.



format-string is a quoted string that specifies how to format the returned character string from the number or time expression. The special symbols that the USING operator recognizes in the *format-string* are described on page 4-92 for number values, and on page 4-94 for DATE values.

Usage

With a number or MONEY operand, you can use the USING operator to align decimal points or currency symbols, to right- or left-justify numbers, to put negative numbers in parentheses, and to perform other formatting tasks. USING can also convert a DATE operand to a variety of formats.

USING is typically used in DISPLAY or PRINT statements, but you can also use it with LET to assign the formatted value to a character variable. If a value is too large for the field, 4GL fills it with asterisks (*) to indicate an overflow.

Formatting Number Expressions

The USING operator takes precedence over the DBMONEY or DBFORMAT environmental variables, and is required to display the *thousands* separator of DBFORMAT. When 4GL displays a number value, it follows these rules:

- Displays the *leading currency symbol* (as set by DBFORMAT or DBMONEY) for MONEY values. If the FORMAT attribute specifies a leading currency symbol for other data types, then 4GL displays that symbol.
- Omits the *thousands* separators, unless they are specified by a FORMAT attribute or by the USING operator.
- Displays the *decimal* separator, except for INT or SMALLINT values.
- Displays the *trailing currency symbol* (as set by DBFORMAT or DBMONEY) for MONEY values, unless you specify a FORMAT attribute or the USING operator. In this case, 4GL ignores the trailing currency symbol; the user cannot enter a trailing currency symbol, and 4GL does not display it.

The USING Formatting Symbols for Number Values

The *format-string* can include literals and special characters * & # < , . - + ( ) $. The following list describes the effects of these special formatting characters:

* This character fills with asterisks (*) any positions in the display field that would otherwise be blank.

& This character fills with zeros any positions in the display field that would otherwise be blank.

This character does not change any blank positions in the display field. You can use this to specify a maximum width for a field.

< This character causes numbers in the field to be left-justified.

, This character is a literal. USING displays it as a comma (but displays no comma unless there is a number to the left of it).

. This character is a literal. USING displays it as a period. You can only have one decimal point (or period) in a number format string.

- This character is a literal. USING displays it as a minus sign when the expression is less than zero, and otherwise as a blank. When you group several minus signs in a row, a single minus sign floats immediately to the left of the number being printed.

+ This character is a literal. USING displays it as a plus sign when the expression is greater than or equal to zero, and as a minus sign when it is less than zero. When you group several plus signs in a row, a single plus sign floats immediately to the left of the displayed number.

(This character is a literal. USING displays it as a left parenthesis before a negative number. It is the *accounting parenthesis* that is used in place of a minus sign to indicate a negative number. When you group several parentheses in a row, a single left parenthesis floats immediately to the left of the number being printed.

) This is the accounting parenthesis that is used in place of a minus sign to indicate a negative number. One of these characters generally closes a format string that begins with a left parenthesis.

$ This character is a literal. USING displays it as a dollar sign. When you group several dollar signs in a row, a single dollar sign floats immediately to the left of the number being printed.

The characters - + ( ) $ *float*, meaning that when you specify multiple leading occurrences of one of these characters, 4GL displays all of them as a single character immediately to the left of the number that is being displayed.

Note: *These are not identical to the formatting characters that you can specify in the format-strings of the* FORMAT (*page 5-42*) *or* PICTURE (*page 5-48*) *field attributes.*

For examples of USING format strings for number expressions, see pages 4-97 through 4-99. Since format strings interact with data to produce visual effects, you may find that the examples are easier to follow than the descriptions on the previous page of USING format string characters.

The following example prints a MONEY value using a format string that allows values up to $9,999,999.99 to be formatted correctly.

```
DEFINE mon_val MONEY(8,2)
LET mon_val = 23485.23
DISPLAY "The current balance is ", mon_val
    USING "$#,###,##&.&&"
```

Executing this DISPLAY statement (with the value of **mon_val** set to 23485.23) produces the following output:

```
The current balance is $   23,485.23
```

The format string in this example fixes the currency symbol.

The format string in the previous example also uses the # and & fill characters. The # character provides blank fill for unused character positions, while the & character provides zero filling. This format ensures that even if the number is zero, any positions marked with & will appear as zero, not blank.

Dollar signs can be used instead of # characters, as in the following statement:

```
DISPLAY "The current balance is ",mon_val
    USING "$$,$$$,$$&.&&"
```

In this example, the currency symbol floats with the size of the number, so that it appears immediately to the left of the most significant digit in the display. This would produce the following formatted output, if the value of the **mon_val** variable were 23485.23:

```
The current balance is    $23,485.23
```

By default, 4GL displays numbers right-justified. You can use the < symbol in a USING format string to override this default. For example, specifying

```
DISPLAY "The current balance is ",mon_val
    USING "$<<,<<<,<<&.&&"
```

produces the following output when the value of **mon_val** is 23485.23:

```
The current balance is $23,485.23
```

Formatting DATE Values

If you use the USING operator to format a DATE value, USING takes precedence over the DBDATE environment variable. The *format-string* for a date can be a combination of the characters *m*, *d*, and *y*:

dd	day of the month as a 2-digit number (01through 31 or less)
ddd	day of the week as a 3-letter abbreviation (Sun through Sat)
mm	month as a 2-digit number (01 through 12)
mmm	month as a 3-letter abbreviation (Jan through Dec)
yy	year as a 2-digit number in the 1900s (00 through 99)
yyyy	year as a 4-digit number (0001 through 9999)

Here lowercase is required; uppercase D, M, or Y cannot be substituted.

Any other characters in a USING format-string for a DATE value are literals. The following examples show valid formats for December 25, 1993:

Format String	Formatted Result
"mmddyy"	122593
"ddmmyy"	251293
"yymmdd"	931225
"yy/mm/dd"	93/12/25
"yy mm dd"	93 12 25
"yy-mm-dd"	93-12-25
"mmm. dd, yyyy"	Dec. 25, 1993
"mmm dd yyyy"	Dec 25 1993
"yyyy dd mm"	1993 25 12
"mmm dd yyyy"	Dec 25 1993
"ddd, mmm. dd, yyyy"	Sat, Dec. 25, 1993
"(ddd) mmm. dd, yyyy"	(Sat) Dec. 25, 1993

The following example is from a REPORT program block:

```
ON LAST ROW
   SKIP 2 LINES
   PRINT "Number of customers in ", state, " are ",
          COUNT(*) USING "<<<<<"
PAGE TRAILER
   PRINT COLUMN 35, "page ", PAGENO USING "<<<<"
```

The following REPORT fragment illustrates several different formats:

```
SKIP 1 LINE
PRINT COLUMN 15, "FROM:  ", begin_date USING "mm/dd/yy",
      COLUMN 35, "TO:  ", end_date USING "mm/dd/yy"
PRINT COLUMN 15, "Report run date:  ",
TODAY USING "mmm dd, yyyy"
   SKIP 2 LINES
PRINT COLUMN 2, "ORDER DATE", COLUMN 15, "COMPANY",
      COLUMN 35, "NAME", COLUMN 57, "NUMBER",
COLUMN 65, "AMOUNT"
BEFORE GROUP OF days
   SKIP 2 LINES
AFTER GROUP OF number
   PRINT COLUMN 2, order_date, COLUMN 15, company CLIPPED,
         COLUMN 35, fname CLIPPED, 1 SPACE, lname CLIPPED,
         COLUMN 55, number USING "####",
         COLUMN 60, GROUP SUM(total_price)
              USING "$$,$$$,$$$.&&"
AFTER GROUP OF days
   SKIP 1 LINE
   PRINT COLUMN 21, "Total amount ordered for the day:  ",
         GROUP SUM(total_price) USING "$$$$,$$$,$$$.&&"
   SKIP 1 LINE
   PRINT COLUMN 15,
         "====================================================="
ON LAST ROW
   SKIP 1 LINE
   PRINT COLUMN 15,
        "======================================================"
   SKIP 2 LINES
   PRINT "Total Amount of orders:  ", SUM(total_price)
         USING "$$$$,$$$,$$$.&&"
PAGE TRAILER
   PRINT COLUMN 28, PAGENO USING "page <<<<"
```

NLS

When NLS is active, the setting in the NLS environment variables LC_NUMERIC and LC_MONETARY affect the way the format string in the USING expression is interpreted for numeric and monetary data. In the format string, the period symbol is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma symbol is a placeholder for the thousands separator specified by environment variables. The $ symbol is a placeholder for the leading currency symbol. The @ symbol is a placeholder for the trailing currency symbol. Thus, the format string $#,###.## formats the value 1234.56 as £1,234.56 in an English locale but as f1.234,56 in a French locale. Note that setting either DBFORMAT or DBMONEY overrides any settings in LC_MONETARY or LC_NUMERIC. For complete information on using NLS, see Appendix E.

The *mmm* and *ddd* specifiers in a format string can display language-specific month name and day name abbreviations. This requires the installation of message files in a subdirectory of **$INFORMIXDIR/msg**, and subsequent reference to that subdirectory by way of the environment variable DBLANG. For example, in a Spanish locale, the *ddd* specifier translates the day Saturday into the day name abbreviation Sab, which stands for Sabado (the Spanish word for Saturday). For more information on NLS, see Appendix E.

USING Operator Examples

Displays on the following pages illustrate capabilities of the USING operator. In these tables, b represents a blank or space.

Format String	Data Value	Formatted Result
"#####"	0	bbbbb
"&&&&&"	0	00000
"$$$$$"	0	bbbb$
"*****"	0	*****
"<<<<<"	0	(*NULL string*)
"<<<,<<<"	12345	12,345
"<<<,<<<"	1234	1,234
"<<<,<<<"	123	123
"<<<,<<<"	12	12
"##,###"	12345	12,345
"##,###"	1234	b1,234
"##,###"	123	bbb123
"##,###"	12	bbbb12
"##,###"	1	bbbbb1
"##,###"	-1	bbbbb1
"##,###"	0	bbbbbb
"&&,&&&"	12345	12,345
"&&,&&&"	1234	01,234
"&&,&&&"	123	000123
"&&,&&&"	12	000012
"&&,&&&"	1	000001
"&&,&&&"	-1	000001
"&&,&&&"	0	000000
"&&,&&&.&&"	12345.67	12,345.67
"&&,&&&.&&"	1234.56	01,234.56
"&&,&&&.&&"	123.45	000123.45
"&&,&&&.&&"	0.01	000000.01
"$$,$$$"	12345	****** (*overflow*)
"$$,$$$"	1234	$1,234
"$$,$$$"	123	bb$123
"$$,$$$"	12	bbb$12
"$$,$$$"	1	bbbb$1
"$$,$$$"	0	bbbbb$
"**,***"	12345	12,345
"**,***"	1234	*1,234
"**,***"	123	***123
"**,***"	12	****12
"**,***"	1	*****1
"**,***"	0	******

Here the character `b` represents a blank or space.

Format String	Data Value	Formatted Result
"##,###.##"	12345.67	12,345.67
"##,###.##"	1234.56	b1,234.56
"##,###.##"	123.45	bbb123.45
"##,###.##"	12.34	bbbb12.34
"##,###.##"	1.23	bbbbb1.23
"##,###.##"	0.12	bbbbb0.12
"##,###.##"	0.01	bbbbbb.01
"##,###.##"	-0.01	bbbbbb.01
"##,###.##"	-1	bbbbb1.00
"$$,$$$.$$"	12345.67	********* (*overflow*)
"$$,$$$.$$"	1234.56	$1,234.56
"$$,$$$.##"	0.00	$.00
"$$,$$$.##"	1234.00	$1,234.00
"$$,$$$.&&"	0.00	$.00
"$$,$$$.&&"	1234.00	$1,234.00
"-$$$,$$$.&&"	-12345.67	-$12,345.67
"-$$$,$$$.&&"	-1234.56	-b$1,234.56
"-$$$,$$$.&&"	-123.45	-bbb$123.45
"--$$,$$$.&&"	-12345.67	-$12,345.67
"--$$,$$$.&&"	-1234.56	-$1,234.56
"--$$,$$$.&&"	-123.45	-bb$123.45
"--$$,$$$.&&"	-12.34	-bbb$12.34
"--$$,$$$.&&"	-1.23	-bbbb$1.23
"-##,###.##"	-12345.67	-12,345.67
"-##,###.##"	-123.45	-bbb123.45
"-##,###.##"	-12.34	-bbbb12.34
"--#,###.##"	-12.34	-bbb12.34
"---,###.##"	-12.34	-bb12.34
"---,-##.##"	-12.34	-12.34
"---,--#.##"	-1.00	-1.00
"-##,###.##"	12345.67	12,345.67
"-##,###.##"	1234.56	1,234.56
"-##,###.##"	123.45	123.45
"-##,###.##"	12.34	12.34
"--#,###.##"	12.34	12.34
"---,###.##"	12.34	12.34
"---,-##.##"	12.34	12.34
"---,---.##"	1.00	1.00
"---,---.--"	-.01	-.01
"---,---.&&"	-.01	-.01

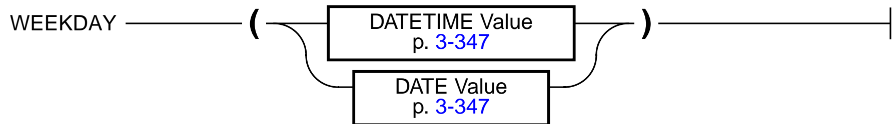
Here the character `b` represents a blank or space.

Format String	Data Value	Formatted Result
"----,--$.&&"	-12345.67	-$12,345.67
"----,--$.&&"	-1234.56	-$1,234.56
"----,--$.&&"	-123.45	-$123.45
"----,--$.&&"	-12.34	-$12.34
"----,--$.&&"	-1.23	-$1.23
"----,--$.&&"	-.12	-$.12
"$***,***.&&"	12345.67	$*12,345.67
"$***,***.&&"	1234.56	$**1,234.56
"$***,***.&&"	123.45	$****123.45
"$***,***.&&"	12.34	$*****12.34
"$***,***.&&"	1.23	$******1.23
"$***,***.&&"	.12	$*******.12
"($$$,$$$.&&)"	-12345.67	($12,345.67)
"($$$,$$$.&&)"	-1234.56	(b$1,234.56)
"($$$,$$$.&&)"	-123.45	(bbb$123.45)
"(($$,$$$.&&)"	-12345.67	($12,345.67)
"(($$,$$$.&&)"	-1234.56	($1,234.56)
"(($$,$$$.&&)"	-123.45	(bb$123.45)
"(($$,$$$.&&)"	-12.34	(bbb$12.34)
"(($$,$$$.&&)"	-1.23	(bbbb$1.23)
"((((,(($.&&)"	-12345.67	($12,345.67)
"((((,(($.&&)"	-1234.56	($1,234.56)
"((((,(($.&&)"	-123.45	($123.45)
"((((,(($.&&)"	-12.34	($12.34)
"((((,(($.&&)"	-1.23	($1.23)
"((((,(($.&&)"	-.12	($.12)
"($$$,$$$.&&)"	12345.67	$12,345.67
"($$$,$$$.&&)"	1234.56	$1,234.56
"($$$,$$$.&&)"	123.45	$123.45
"(($$,$$$.&&)"	12345.67	$12,345.67
"(($$,$$$.&&)"	1234.56	$1,234.56
"(($$,$$$.&&)"	123.45	$123.45
"(($$,$$$.&&)"	12.34	$12.34
"(($$,$$$.&&)"	1.23	$1.23
"((((,(($.&&)"	12345.67	$12,345.67
"((((,(($.&&)"	1234.56	$1,234.56
"((((,(($.&&)"	123.45	$123.45
"((((,(($.&&)"	12.34	$12.34
"((((,(($.&&)"	1.23	$1.23
"((((,(($.&&)"	.12	$.12

Here the character `b` represents a blank or space.

WEEKDAY( )

The WEEKDAY( ) operator returns a positive integer, corresponding to the day of the week implied by its DATE or DATETIME operand.



Usage

WEEKDAY( ) accepts a DATETIME or DATE operand, and returns an integer in the range `0` through `6`. Here zero represents Sunday, `1` represents Monday, and so on.

The following example demonstrates a function called from inside a FOR loop. This function makes use of WEEKDAY, together with a CASE statement to assign a three-letter day of the week abbreviation to each date in an array, skipping weekends.

```
FOR i = 1 TO 10
   CALL seize_theday(next_day)
      RETURNING day_name, next_day
   LET pa_days[i].dayo_week = day_name
   LET pa_days[i].rdate = next_day
   LET next_day = next_day + 1
END FOR
...
FUNCTION seize_theday(next_day)
   DEFINE
      week_day SMALLINT
      day_name CHAR(3)
      next_day DATE
   LET week_day = WEEKDAY(next_day)
   CASE week_day
         WHEN 1
            LET day_name = "Mon"
         WHEN 2
            LET day_name = "Tues"
         WHEN 3
            LET day_name = "Wed"
         WHEN 4
            LET day_name = "Thu"
```

```
            WHEN 5
               LET day_name = "Fri"
            WHEN 6
               LET day_name = "Mon"
               LET next_day = next_day + 2
            WHEN 7
               LET day_name = "Mon"
               LET next_day = next_day + 1
         END CASE
      RETURN day_name, next_day
   END FUNCTION -- seize_theday
```

This operator is useful for determining the day of the week from dates in recent and future centuries. It should be used with caution, however, for more remote dates, because of disagreements among the various calendar systems used in different countries before the Gregorian year 1750.

For dates thousands of years in the past (for example, the death of Socrates), it is difficult to verify that the sequential count of the seven days of the week has been accurately maintained from antiquity up to the present.

The WEEKDAY( ) operator is among a group of built-in 4GL operators that extract a single time unit value from a DATETIME or DATE value. These are the "extraction" operators of 4GL that accepts a DATETIME or DATE operand:

Operator	Meaning of the Returned Integer
DAY( )	The day of the month
MONTH( )	The month
YEAR( )	The year
WEEKDAY( )	The day of the week

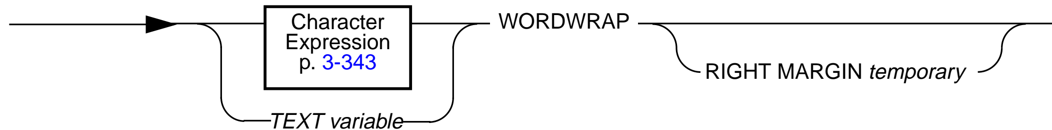
In addition to these, the DATE( ) operator can extract the date portion of a DATETIME value that has YEAR TO DAY or greater precision.

References

DATE( ), DAY( ), MONTH( ), YEAR( )

WORDWRAP

The WORDWRAP operator divides a long text string into segments that appear in successive lines of a 4GL report. (This operator can appear only in the PRINT statement in the FORMAT section of a REPORT program block.)



temporary is an integer expression (page 3-338) whose returned value specifies the absolute position (in characters), counting from the left edge of the page, of a temporary right margin.

TEXT variable is the name of a **4GL** variable of the TEXT data type.

Usage

The WORDWRAP operator automatically "wraps" successive segments of long character strings onto successive lines of output from a 4GL report. The string value of any expression or *TEXT variable* that is too long to fit between the current character position and the specified or default right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary *left margin*.
- Unless you specify RIGHT MARGIN *temporary*, the *right margin* defaults to 132, or to the *size* from the RIGHT MARGIN clause of the OUTPUT section.

Specify WORDWRAP RIGHT MARGIN *integer expression* to set a temporary right margin, counting from the left edge of the page. This cannot be smaller than the current character position, nor greater than 132 (or else the *size* from the RIGHT MARGIN clause of the OUTPUT section). The current character position becomes the temporary left margin. These temporary values override the specified or default left and right margins from the OUTPUT section.

After the PRINT statement has executed, any explicit or default margins from the OUTPUT section are restored. For more information about the PRINT statement in 4GL reports, see page 6-42.

The following PRINT statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the 4GL variable called **mynovel**:

```
print column 10, mynovel WORDWRAP RIGHT MARGIN 70
```

Tabs, Line Breaks, and Page Breaks with WORDWRAP

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), Newline (ASCII 10), and RETURN (ASCII 13) characters that partition the string into "words," consisting of substrings of other printable characters. Other non-printable characters may cause runtime errors. If the data string cannot fit between the margins of the current line, **4GL** breaks the line at a *word* division, padding the line with blanks at the right.

From left to right, **4GL** expands any TAB character to enough blank spaces to reach the next TAB stop. By default, TAB stops are in every eighth column, beginning at the left-hand edge of the page. If the next TAB stop or a string of blank characters extends beyond the right margin, **4GL** takes these actions:

- Print blank characters only to the right margin.
- Discard any remaining blank characters from the blank string or TAB.
- Start a new line at the temporary left margin.
- Process the next word.

4GL starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, this creates an even left margin. **4GL** applies the following rules (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any Newline, or RETURN, or Newline and RETURN pair.
- Break at the last blank (ASCII 32) or TAB character before the right margin.
- Break at the right margin, if no character farther to the left is a blank, RETURN, TAB, or Newline character.

4GL maintains page discipline under the WORDWRAP option. If the string is too long for the current page, **4GL** executes the statements in any page trailer and header control blocks before continuing output onto a new page.

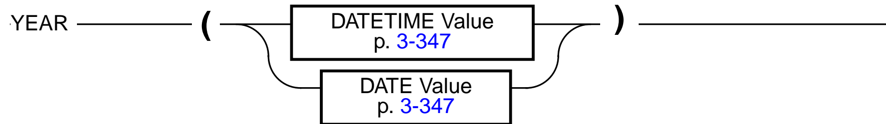
Note: *The* WORDWRAP *keyword can also specify a field attribute (page 5-57) that supports data display and data entry in a multiple-segment field of a 4GL form.*

References

LINENO, PAGENO, SPACES

YEAR( )

The YEAR( ) operator returns an integer, corresponding to the *year* portion of its DATE or DATETIME operand.



Usage

The YEAR( ) operator returns all the digits of the year value (`1994`, not `94`). (See, however, the second example below, which illustrates how to obtain a two-digit value like `94` from a four-digit year like `1994`.)

The following example extracts the current year and stores the value in a variable.

```
LET y_var = YEAR(TODAY)
```

You can produce a two-digit year abbreviation by using the MOD (modulus) operator:

```
LET birth_yr = (YEAR(birth_date)) MOD 100
```

In the right-hand expression, the MOD operator yields the year modulo 100, the remainder when the value representing the actual year is divided by 100.

For example, if the value of **birth_date** is `09-16-1953`, the YEAR( ) operator extracts the value `1953`, and the expression `1953 MOD 100` returns `53`. That value is then assigned to **birth_yr**.

References

DATE( ), DAY( ), MONTH( ), WEEKDAY( )

Chapter

5

Screen Forms

4GL Forms 3
- Form Drivers 3
- Form Fields 4
  - Appearance of Fields 5
  - Navigation Among Form Fields 5
  - Disabled Form Fields 6

Structure of a Form Specification File 6

DATABASE Section 10
- Database References in the DATABASE Section 11
- The FORMONLY Option 11
- The WITHOUT NULL INPUT Option 12

SCREEN Section 12
- The SIZE Option 13
- The Screen Layout 14
- Display Fields 14
- Literal Characters in Forms 15
  - Graphics Characters in Forms 16
  - Rectangles Within Forms 18

TABLES Section 18
- Table Aliases 19

ATTRIBUTES Section 20
- Fields Linked to Database Columns 21
- FORMONLY Fields 24
  - The Data Type Specification 25
  - The NOT NULL Keywords 25
- Multiple-Segment Fields 26
- Field Attributes 27
- Field Attribute Syntax 28
- AUTONEXT 30

COLOR 31
Boolean Expressions in 4GL Form Specification Files 32
COMMENTS 36
DEFAULT 38
DISPLAY LIKE 40
DOWNSHIFT 41
FORMAT 42
INCLUDE 44
INVISIBLE 46
NOENTRY 47
PICTURE 48
PROGRAM 50
REQUIRED 52
REVERSE 53
UPSHIFT 54
VALIDATE LIKE 55
VERIFY 56
WORDWRAP 57

INSTRUCTIONS Section 63
Screen Records 63
Non-Default Screen Records 64
The List of Member Fields 65
Screen Arrays 66
Field Delimiters 68

Default Attributes 69
Precedence of Field Attribute Specifications 72
Default Attributes in an ANSI-Compliant Database 72

Creating and Compiling a Form 73
Compiling a Form Through the Programmer's Environment 73
Compiling a Form Through the Operating System 74
Default Forms 75

Using PERFORM Forms in 4GL 77

4GL Forms

A *screen form* is a visual display that can support input or output tasks in an INFORMIX-4GL application. Before your 4GL program can use a screen form, you must first create a *form specification file*. This is an ASCII source file that describes the logical format of the screen form, and how to display data values in the form at runtime. It must be compiled separately from the rest of your source code. A compiled form can be used by many 4GL programs.

Most of this chapter describes the structure of a form specification file, and the effect and syntax of its components. The section "Default Attributes" on page 5-69 describes the **syscolval** and **syscolatt** tables that can specify formats, validation rules, and default data values for fields. For information on using the **upscol** utility to specify values in these tables, see Appendix B. Additional sections in this chapter describe how to compile 4GL forms and how to use forms designed for PERFORM, the screen transaction processor of INFORMIX-SQL.

Form Drivers

To work with a compiled screen form, the application requires a *form driver,* a logical set of 4GL statements that control the display of the form, that binds its fields to 4GL variables, and that respond to actions by the user in fields. The form driver can include 4GL screen interaction and data manipulation statements to enter, retrieve, modify, or delete data in the database. The emphasis of this chapter, however, is on how to create the form specification file, rather than on how to design and implement the form driver.

Regardless of how you define them, there is no implicit relationship between the values of program *variables*, form *fields*, and database *columns*. Even, for example, if a 4GL variable **lname** is declared LIKE **customer.lname**, changes to the variable do not imply any change in the column value. Relationships among these entities must be specified in the logic of your form driver.

Similarly, a 4GL form is only a template. **FORM4GL** reads the system catalog at compile time to obtain the names and data types of any columns that are referenced in the form specification file. After compilation, however, the form loses its connection to the database. It can no longer distinguish the name of a table or view from the name of a screen record.

It is up to you, the programmer, to determine what data a form displays, and what to do with data that the user enters into the fields of a form. You must indicate the binding explicitly in any 4GL statement that connects 4GL variables to screen forms or to database columns. The following statements, for example, take input from a 4GL form, and insert the entered value from the form into the database. (Here the @ sign in the INSERT statement tells 4GL that the first **lname** is the SQL identifier of a database column.)

```
INPUT lname FROM customer.lname
INSERT INTO customer (@lname) VALUES (lname)
```

You can use interactive 4GL statements like OPEN FORM, OPEN WINDOW, INPUT, DISPLAY FORM, CLEAR FORM, and CONSTRUCT in the form driver to support data entry or data display through the 4GL form. Some statements support temporary binding when a program variable and a screen field have identical names. (See the individual 4GL statement descriptions in Chapter 3 for the appropriate syntax.) For example, the following statement could replace the previous INPUT statement:

```
INPUT BY NAME lname
```

For more information about form drivers, refer to the later chapters of *INFORMIX-4GL Concepts and Use*.

Form Fields

In a 4GL form, a *field* (sometimes called a *screen field* or *form field*) is an area where the user of the application can view, enter, or edit data, depending on its description in the form specification file and statements in the form driver. This section discusses the appearance and behavior of form fields in 4GL.

Appearance of Fields

The screen form contains *display fields* bounded by *delimiters* such as the square brackets shown below.

```
- - - - - - - - - - - - - - - - - - - - - - - -
                      CUSTOMER  FORM

      Number:  [           ]

  First Name:  [               ]     Last Name:  [                ]

     Company:  [                     ]

     Address:  [                     ]
               [                     ]

        City:  [                ]

       State:  [  ]   Zipcode:  [      ]

   Telephone:  [                   ]
- - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 5-1 ***The* customer *Form***

The currently active form field contains a cursor. This is where text that you type appears. For details of how to size and position fields in a form, see "Display Fields" on page 5-14. For information on assigning display and validation attributes to fields, see "Field Attribute Syntax" on page 5-28.

Navigation Among Form Fields

The order in which the cursor moves from field to field on a screen form is determined by the order in which you list fields in the INPUT statement. Any time before pressing RETURN in the last field, the user can use the Arrow keys to move back through the fields and make corrections. The user can indicate that data entry is complete by pressing the Accept key in any field, or by pressing RETURN in the last field.

The FIELD ORDER setting in the OPTIONS statement determines where the Arrow keys move the cursor. If FIELD ORDER CONSTRAINED is specified, the Up Arrow key moves the cursor to the previous field and the Down Arrow key moves the cursor to the next field. If FIELD ORDER UNCONSTRAINED is

specified, the Up Arrow key moves the cursor to the field above the current cursor position and the Down Arrow key moves the cursor to the field below the current cursor position.

Disabled Form Fields

When a form field is not included in a screen interaction statement like INPUT or CONSTRUCT, or is specified as a NOENTRY field in a form file, it is disabled during execution of that statement. The user cannot move the cursor to the field. If the user attempts to enter the NOENTRY field by using the TAB or Arrow keys, the cursor moves to the next field in traversal order, and any appropriate BEFORE and AFTER clauses are executed.

Form fields are also disabled prior to execution of a screen interaction statement like INPUT or CONSTRUCT. Data entered into the fields prior to the execution of the interactive statement would otherwise be lost when the statement initializes the fields.

Structure of a Form Specification File

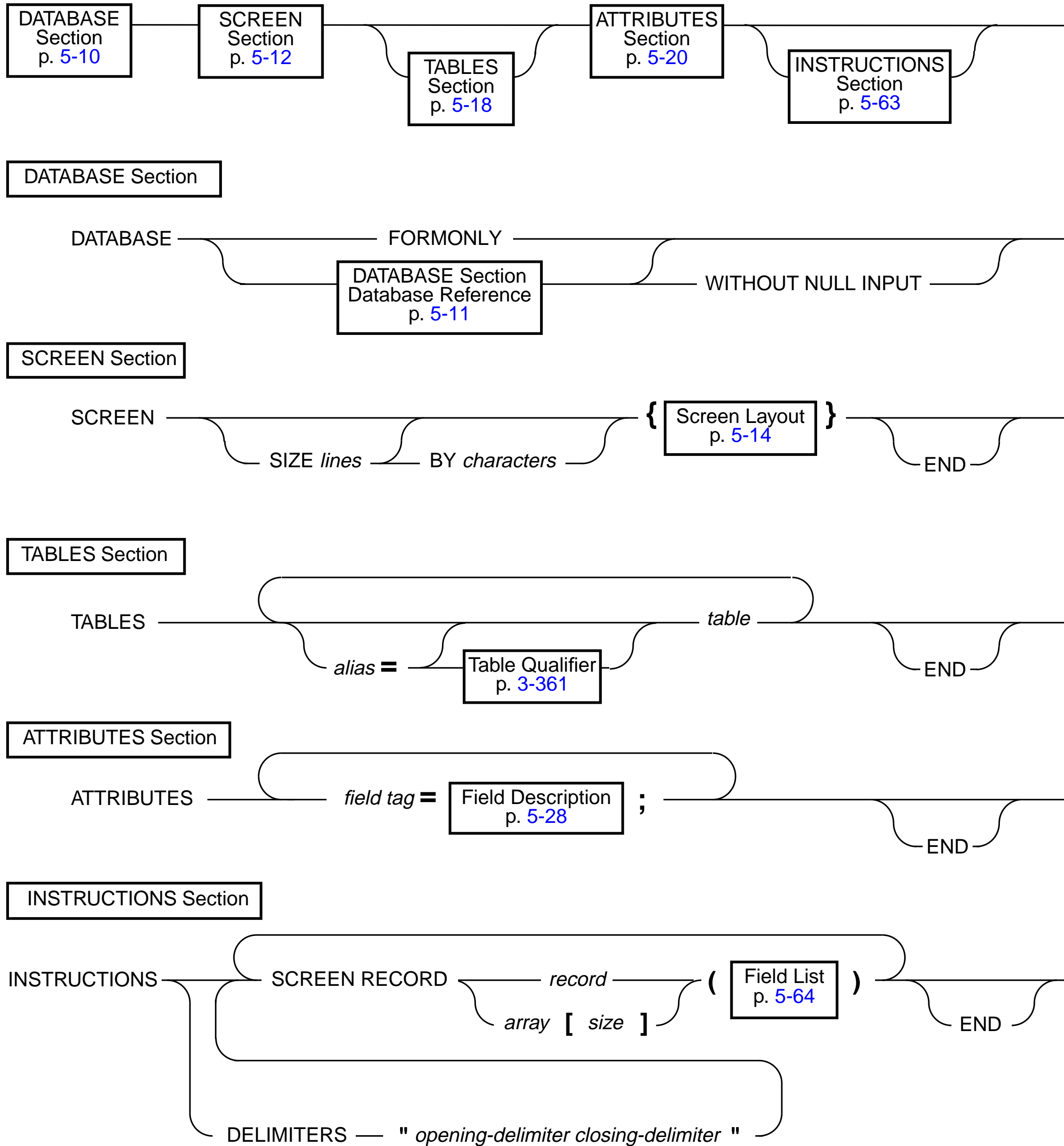
A 4GL form specification file is an ASCII file (with file extension **.per**) that you can create with a text editor or from within the 4GL Programmer's Environment. This file consists of three required sections (DATABASE, SCREEN, and ATTRIBUTES) and it can also include two optional sections (TABLES and INSTRUCTIONS). If present, these five sections must appear in the following order:

- **DATABASE Section**: Each form specification file must begin with a DATABASE section identifying the database (if any) on which the form is based. This can be any database that your database engine can access, including a remote database.
- **SCREEN Section**: The SCREEN section must appear next, showing the dimensions and the exact layout of the logical elements of the form. You must specify the position of one or more *screen fields* for data entry or display, and any additional text or ornamental characters.
- **TABLES Section**: The TABLES section must follow the SCREEN section of any form that references the identifier of a database column. This section lists every table or view that contains columns that are referenced in the ATTRIBUTES or INSTRUCTIONS sections. The TABLES section must also declare an *alias* for any table name or synonym that requires an *owner* qualifier, or that is an external table, or an external, distributed table.

- **ATTRIBUTES Section**: The ATTRIBUTES section describes each field on the form and assigns names to fields. Field descriptions can optionally include *field attribute*s to specify, for example, the appearance, acceptable input values, on-screen comments, and default values for each field.
- **INSTRUCTIONS Section**: The INSTRUCTIONS section is optional. It can specify screen arrays, and non-default screen records.

Each section must begin with the keyword for which it is named. After you create a form specification file, you must compile it. The form driver of your 4GL application can then use *4GL* variables to transfer information between the database and the fields of the screen form.

This is the syntax of a **4GL** form specification:



The next five sections of this chapter identify the keywords and terms listed in this diagram, and describe their syntax in detail.

Figure 5-2 illustrates the overall structure of a form specification file:

```
DATABASE stores

SCREEN
{
----------------------------------------------------------------------
CUSTOMER INFORMATION:
Customer Number: [c1         ]             Telephone: [c10          ]
           ...

SHIPPING INFORMATION:
     Customer P.O.: [o20       ]

         Ship Date: [o21       ]          Date Paid: [o22       ]
}

TABLES
customer orders items manufact

ATTRIBUTES
c1 = customer.customer_num
   = orders.customer_num;
c10 = customer.phone, PICTURE = "###-###-####x#####";
           ...
o20 = orders.po_num;
o21 = orders.ship_date;
o22 = orders.paid_date;

INSTRUCTIONS
SCREEN RECORD sc_order[5] (orders.order_date THRU orders.paid_date)
```

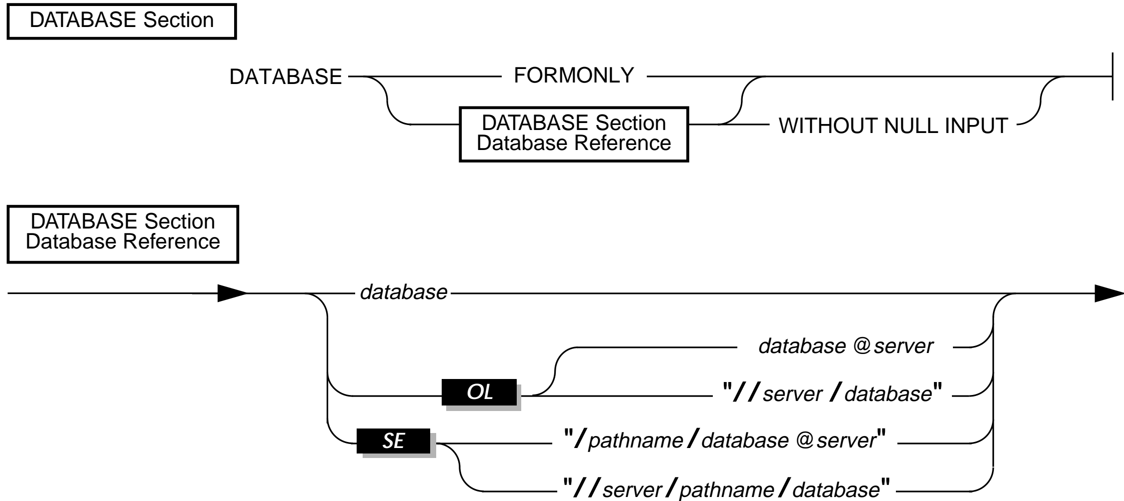
Figure 5-2 Sections of a form specification file

In this example, the screen form has been designed to display columns from several tables in the stores demonstration database, and includes all five of the required and optional sections that are described in the pages that follow.

This example is incomplete, since it omits portions of the SCREEN and ATTRIBUTES sections that describe some of the screen fields. The ellipsis notation (. . .) in those sections is a typographic device to simplify this illustration, rather than a valid specification for the form.

DATABASE Section

The DATABASE section identifies the database, if any, containing tables or views whose columns are referenced in the form specification file.



database is the SQL identifier of a database (page 3-58).

pathname is the directory path to the parent directory of the **.dbs** directory.

server is the name of the host system where *database* resides.

Usage

The DATABASE section is required, even if the screen form does not reference any database columns or tables. You can specify no more than one database.

When compiling forms, **4GL** uses the schema of tables from the specified database to define the data types of fields in the form, and obtains default values and attributes from the **syscolval** and **syscolatt** tables in the default database.

Database References in the DATABASE Section

If the form specification file references any table or column names from a database, the DATABASE section must specify exactly one *database reference*. For the INFORMIX-OnLine engine, the following are all valid formats:

database
database@server
"*//server/database*"

The last format requires single quotation marks.

The next examples specify that columns or tables referenced in the TABLES, ATTRIBUTES, or INSTRUCTIONS sections are in the **stores2** database; the last two DATABASE section examples specify **mammoth** as the OnLine server:

```
DATABASE stores2
DATABASE stores2@mammoth
DATABASE "//mammoth/stores2"
```

For databases supported by the INFORMIX-SE database engine, the following are all valid formats:

database
"*/pathname/database@server*"
"*//server/pathname/database*"

Single quotation marks around the last two formats are mandatory. Here *pathname* is a pathname to the directory that contains *database*, and *server* is the name of the host system where *database* resides.

For INFORMIX-SE, the following DATABASE sections illustrate these formats:

```
DATABASE newdb
DATABASE "/usr/projects/newdb@mammoth"
DATABASE "//mammoth/usr/projects/newdb"
```

The FORMONLY Option

It is possible to create a form that is not related to any database. To do so, specify FORMONLY after the DATABASE keyword, and omit the TABLES section (page 5-18). Also specify FORMONLY as the only table name in the ATTRIBUTES section when you declare the name of each field (page 5-20).

The following example of a DATABASE section specifies that the screen form is not associated with any database:

```
DATABASE FORMONLY
```

Compilation errors may result if FORMONLY appears in the DATABASE section of a form that also specifies features that depend on information from the system catalog or from the **syscolval** and **syscolatt** tables of a database. Features of 4GL forms that depend on a database include:

- The TABLES section.
- Any field associated with a database column in the ATTRIBUTES section.
- Any FORMONLY field declared LIKE a column in the ATTRIBUTES section.
- DISPLAY LIKE or VALIDATE LIKE attributes in the ATTRIBUTES section.

The WITHOUT NULL INPUT Option

The WITHOUT NULL INPUT keywords indicate that *database-name* does not support NULL values. Use this option only if you have elected to create and work with a database that does not have NULL values.

For fields that have no other defaults, the WITHOUT NULL INPUT option causes the form to display *zeros* as default values for number and INTERVAL fields, and *blanks* for character fields. DATE values default to `12/31/1899`. The default value for DATETIME fields is `1899-12-31 23:59:59.99999`.

SCREEN Section

The SCREEN section of the form specification file specifies the vertical and horizontal dimensions of the physical screen, and the position of one or more display fields and other information that will appear on the screen form. This section is required. It has the following syntax:

lines is a literal integer that specifies how many lines of characters (measured vertically) the form can display. The default is 24.

characters is a literal integer, specifying how many characters (measured horizontally) a line can display. The default is the maximum number of characters in any line of the screen layout (page 5-14).

Usage

The SCREEN keyword is required. As in other sections of a form specification, the keyword END is optional.

A single pair of braces ({ }) symbols, immediately preceded and immediately followed by Newline characters, must enclose the *screen-layout*. Do not use braces ({ }) as comment indicators within the SCREEN section.

The SIZE Option

If you omit the SIZE keyword, *lines* defaults to 24, and *characters* defaults to the maximum number of characters in any line of your *screen-layout*. If you specify a default form from within the Programmer's Environment (as described on page 5-75), the SIZE default values appear explicitly in the file.

Specify *lines* as the total height of the form. Four lines are reserved for the system, so no more than (*lines* - 4) lines of the form can display data.

If the value of (*lines* - 4) is less than the number of lines in the *screen-layout*, then **FORM4GL** splits your form into a new page after every (*lines* - 4) lines. 4GL does not support multiple-page forms, so any lines beyond the first page will overlay the last line of the first page if your *screen-layout* is too large for your screen. (To avoid this superimposition, you should create several form specification files if you need to display more lines than can fit on one form.)

The **form4gl** command can override either or both of the *lines* or the *characters* dimensions of the SCREEN section by specifying the following:

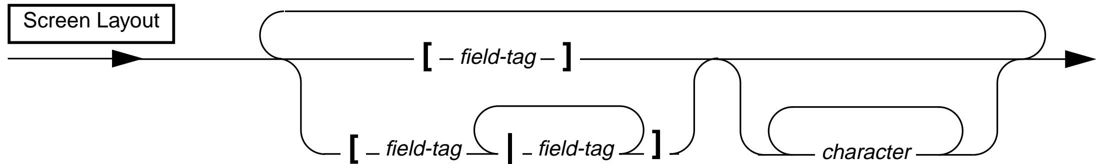
```
form4gl -l lines -c characters form-name
```

where *lines* is the height of the screen, *characters* is the width of the screen, and *form-name* is the filename (without the **.per** extension) of a form specification file. For complete information on the **form4gl** command, see "Compiling a Form Through the Operating System" on page 5-74.

The portion of the SCREEN section between the braces symbols is called the *screen-layout*. This shows the geometric arrangement of the logical screen. If the SIZE clause or command line specifies dimensions that are too small for the screen-layout, then **FORM4GL** issues a compile-time warning, but it produces the compiled form that your form specification file described.

The Screen Layout

The screen layout of the SCREEN section must be enclosed between a pair of braces ({ }), each in the first character position of an otherwise empty line. The screen layout consists of *display fields* and (optionally) *text characters.*



field-tag is a 4GL identifier of no more than 50 characters within each field. The length of a field tag cannot exceed the field width.

character is a printable character of text that will appear in the form.

Display Fields

Every 4GL form must include at least one *field* where data can appear in the form. Use bracket ([ ]) symbols to delimit these fields in the screen layout. Between delimiters, each field must have an identifying field tag (page 5-15).

Field Delimiters

Each field must be indicated by left and right delimiters to show the length of the field and its position within the screen layout. Both delimiters must appear on the same line. Usually you use left and right brackets to delimit fields. However, if you want two fields to appear directly next to each other, you can use the vertical bar to indicate the end of the first field and the beginning of the second field. For complete information on using a vertical bar (|) to delimit fields, see “Field Delimiters” on page 5-68.

Field Length

If you create a non-default form, you normally should set the width of each display field in the SCREEN section to be equal to the width of the program variable or the database column to which it corresponds.

A field corresponding to a number column in the database should be large enough to contain the largest value in that column. If the field is too small to display an assigned number, 4GL fills the field with asterisk (*) symbols to indicate the overflow.

Fields intended to display character data can be shorter than the declared length. INFORMIX-4GL fills a field from the left, and truncates from the right any character string that is longer than the field to which it is assigned. By using multiple-segment fields (page 5-26), you can display portions of a long character value in successive lines of the form.

In a default form specification file, the widths of all fields are determined by the data type of the corresponding columns in the database tables. (See the section "Creating and Compiling a Form" on page 5-73 for more information about default forms.) Default field widths are listed on page 5-75 for each data type.

If you edit and modify the default form specification file or create a new file, you can verify that the field widths match the data length requirements of the corresponding character columns when you compile the form. For information on compiling a form at the command line, see "Compiling a Form Through the Operating System" on page 5-74.

Field Tags

Field tags must follow the rules for 4GL identifiers (page 2-9). The first character of a *field tag* must be a letter or underscore (_) symbol. Other characters can be any combination of letters, digits, and underscores. Because **FORM4GL** is not case-sensitive, both **a1** and **Al** represent the same field tag. Field tags cannot be referenced in 4GL statements. The ATTRIBUTES section (page 5-20) declares a *field name* for each field tag.

The same field tag can appear at more than one position in the SCREEN section only as part of a multiple-segment field (page 5-26), or as part of a screen array (page 5-66). Otherwise, each field tag must be unique within a form.

You can give single-character fields the tags **a** through **z** (so a form can include no more than 26 single-character fields.)

Literal Characters in Forms

A screen layout can specify strings of ASCII characters that always appear in the screen form. These characters can label the form and its fields, or otherwise improve the display. Text cannot overlap display fields, but the PICTURE attribute (described in "Field Attribute Syntax" on page 5-28) can specify literal characters within CHAR fields.

The SCREEN section listed below appears in the **orderform.per** form specification file in the stores demonstration application. This uses default screen dimensions (24 by 80). Notice the use of textual information for field labels, a

screen title, and ornamental lines. (The “INSTRUCTIONS Section” later in this chapter describes how repeated field tags are used in forms that define screen arrays.)

```
SCREEN
{
-------------------------------------------------------------------------------
                              ORDER FORM
-------------------------------------------------------------------------------
Customer Number:[f000       ]  Contact Name:[f001           ][f002           ]
   Company Name:[f003                ]
        Address:[f004                ][f005                 ]
           City:[f006           ] State:[a0] Zip Code:[f007 ]
      Telephone:[f008              ]
-------------------------------------------------------------------------------
Order No:[f009       ]  Order Date:[f010      ]  Purchase Order No:[f011      ]
    Shipping Instructions:[f012                                    ]
-------------------------------------------------------------------------------
Item No.  Stock No.  Code    Description    Quantity    Price       Total
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018       ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018       ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018       ]
[f013  ]  [f014  ]  [a1 ] [f015           ] [f016  ] [f017     ] [f018       ]
                Running Total including Tax and Shipping Charges:[f019       ]
===============================================================================
}
END
```

Note: *The backslash (\ ) symbol is not valid as a text character;* **FORM4GL** *attempts to interpret it as the beginning of an escape sequence, and does not print it. In addition, your form may not compile correctly if you attempt to use either braces (* { *and* } *) or the field delimiter (* [,] *, and* | *) symbols as text characters in the screen layout.* **FORM4GL** *interprets any pound (* # *) sign or double-hyphen (* -- *) symbols in the screen layout as literals, not as comment indicators.*

Graphics Characters in Forms

You can include graphics characters in the SCREEN section to place boxes and other rectangular shapes in a screen form. Use the following characters to indicate the borders of one or more boxes on the form:

Symbol	Purpose
p	Use p to mark the upper-left corner.
q	Use q to mark the upper-right corner.
b	Use b to mark the lower-left corner.
d	Use d to mark the lower-right corner.
-	Use hyphens (-) to indicate horizontal line segments.
		Use vertical (	) bars to indicate vertical line segments.

The meanings of these six special characters are derived from the **gb** or **acsc** specifications in the **termcap** or **terminfo** files, respectively. **INFORMIX-4GL** substitutes the corresponding graphics characters when you display the compiled form.

Once the form has the desired configuration, use the **\g** string to indicate when to begin graphics mode and when to end graphics mode.

Insert a **\g** string before the first **p**, **q**, **d**, **b**, hyphen, or vertical bar that represents a graphics character. To leave graphics mode, insert the string **\g** after the **p**, **q**, **d**, **b**, hyphen, or vertical bar.

Do not insert a **\g** string into original white space of a screen layout. The backslash should displace the first graphics character in the line, and push the remaining characters to the right. The process of indicating graphics distorts the appearance of a screen layout in the SCREEN section, compared to the corresponding display of the screen form.

You can include other graphics characters in a form specification file. The meaning, however, of a character other than the **p**, **q**, **d**, **b**, hyphen, and vertical bar is terminal-dependent.

To use graphics characters, the system **termcap** or **terminfo** files must include entries for the following variables:

termcap:

gs the escape sequence for entering graphics mode.

ge the escape sequence for leaving graphics mode.

gb the concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border.

terminfo:

smacs the escape sequence for entering graphics mode.

rmacs the escape sequence for leaving graphics mode.

acsc the concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border.

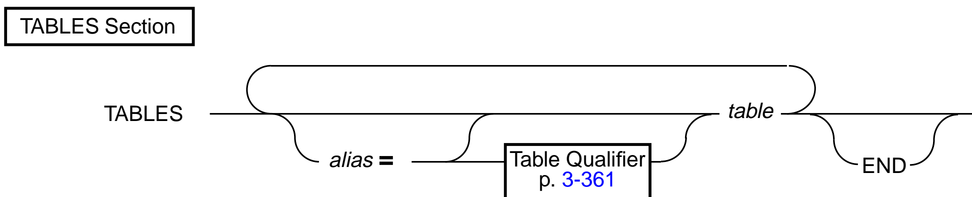
See Appendix F and the manual that comes with your terminal for information about making changes to your **termcap** or **terminfo** files to support these graphics characters.

Rectangles Within Forms

You can use the built-in FGL_DRAWBOX( ) function (page 4-56) to enclose parts of the screen layout within rectangles. Rectangles that you draw with FGL_DRAWBOX( ) are part of a displayed form. Each time that you execute the corresponding DISPLAY FORM or OPEN WINDOW ... WITH FORM statement, you must also redraw the rectangle. Avoid having the rectangle intersect any field or any 4GL reserved line, because the rectangle will be broken at the intersection when anything is displayed in the field or in the reserved line.

TABLES Section

The TABLES section lists the database tables that are referenced elsewhere in the form specification file. You must list in this section any table, view, or synonym that includes a column whose name is referenced in the form.



alias is the alias that replaces *table* in the form specification file.

table is the identifier or synonym of a table or view in its database.

Usage

If the DATABASE section specifies FORMONLY, no TABLES section is needed, unless you give a field the VALIDATE LIKE or DISPLAY LIKE attribute in the ATTRIBUTES section, or specify a FORMONLY field LIKE a database column.

Every database column referenced in the ATTRIBUTES section must be part of some *table* specified in the TABLES section. The *table* identifier is the name listed in the **tabname** column of the **systables** catalog, or else a synonym.

4GL allows you to specify up to 20 *tables*, but the actual limit on the number of tables, views, and synonyms that you can reference in a form depends on how your system is configured. The form specification file ORDERFORM.PER in the demonstration application lists four tables:

```
TABLES   customer   orders   items   stock
```

The *table* cannot be a *temporary* table. If the form supports entry or update of data in a view, then your 4GL application should test at run time whether the view is updatable, especially if it is based upon other views.

The END keyword is optional.

Table Aliases

The TABLES section must declare an *alias* for the identifier of any table, view, or synonym that requires a table qualifier (page 3-361). Table qualifiers can specify the *owner* of a table, or a *database* (or *database@server*) that is different from the database in the DATABASE section. In an ANSI-compliant database, for example, you must qualify any table name with the *owner* prefix if the form will be run by users other than *owner.* You do not need to specify *alias*, unless the form will be used in an ANSI-compliant database by a user who did not create *table*, or if the form references a table, view, or synonym whose name is the same as another in the same database, so that the *owner* prefix is required for an unambiguous reference.

The alias can be the same identifier as *table.* For example, **stock** can be the alias for **stores@naval:tom.stock**. Except to assign an *alias* in the TABLES section, a form specification file cannot qualify the name of a table. If a qualifier is needed, you must use an alias from the TABLES section to reference the table in other sections of the form specification file.

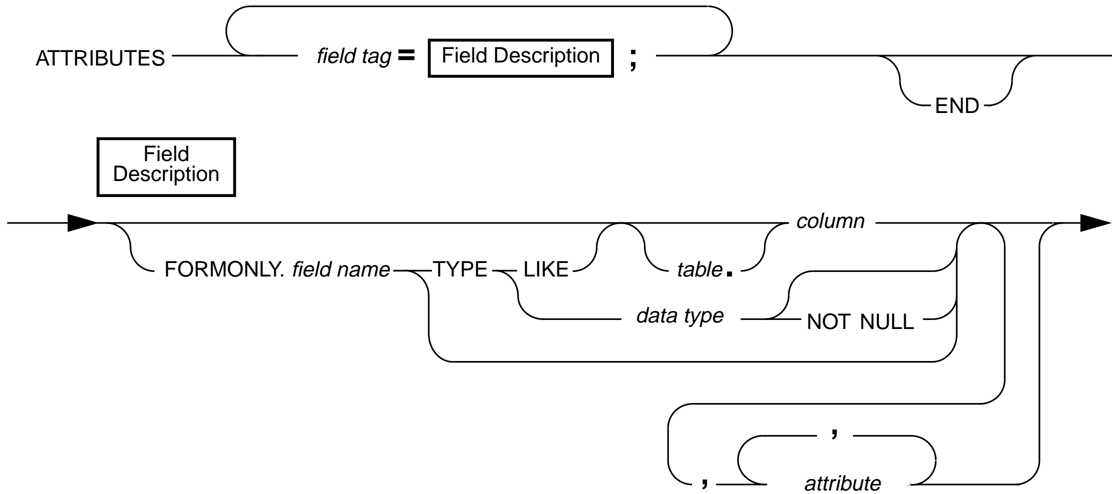
The same alias must also appear in screen interaction statements of 4GL that reference screen fields that are linked to columns of a table that has an alias. Statements in **4GL** programs or in other sections of the form specification file can reference screen fields as *column*, as *alias. column*, or as *table. column*, but they cannot specify *owner. table. column.* You cannot specify *table. column* as a field name if you define a different *alias* for *table.*

The following TABLES section specifies aliases for two tables:

```
TABLES  tab1  =  libdpt.booktab
        tab2  =  athdpt.balltab
```

ATTRIBUTES Section

The ATTRIBUTES section specifies a *field description* that associates an *identifier* and a *data type* with every field in the SCREEN section. You can also control the behavior and appearance of each field by using *field attributes* to describe how INFORMIX-4GL should display the field, supply a default value, limit the values that can be entered, or set other parameters. Field attributes are described in the section "Field Attribute Syntax" on page 5-28.



attribute is a string of keywords, identifiers, and symbols to specify a field attribute, from among those listed on page 5-28.

column is the unqualified SQL identifier of a database column.

data type is any data type specification (page 3-293) except ARRAY or RECORD.

field name is an identifier that you assign to a FORMONLY field (a field that is not associated here with any database column).

field-tag is the field tag, as declared in the SCREEN section.

table is the name or alias of a database table, synonym, or view, as declared in the TABLES section. This is not required, unless several columns in different tables have the same name, or the table is an external table.

Usage

The ATTRIBUTES section must describe every *field-tag* from the SCREEN section. The order in which you list the field tags determines the order of fields in the default screen records that INFORMIX-4GL creates for each table. (The INSTRUCTIONS section on page 5-63 describes screen records.)

The END keyword is optional. It is supported to provide compatibility with form specification files for earlier versions of Informix products.

You can specify two kinds of field descriptions: those that associate a field tag with the data type and with the default attributes of a database column, and those that link field tags to FORMONLY fields.

Fields Linked to Database Columns

Unless a display field is FORMONLY, its *field-description* must specify the SQL identifier of some database column as the name of the display field. Fields are associated with database columns only during the compilation of the form specification file. During the compilation process, **FORM4GL** examines two optional tables, **syscolval** and **syscolatt**, for default values of the attributes that you have associated with any columns of the database. (For a description of these tables, see "Default Attributes" on page 5-69.)

After **FORM4GL** extracts any default attributes and identifies data types from the system catalog, the association between fields and database columns is broken, and the form cannot distinguish the name of a table or view from the name of a screen record. The form driver in your 4GL program must mediate between screen fields and database columns with program variables.

```
field-tag = [table .] column [ , attribute [ , attribute ... ] ] ;
```

attribute is a string of keywords, identifiers, and symbols to specify a field attribute, from among those listed on page 5-28.

column is the unqualified SQL identifier of a database column. This can also appear in 4GL statements that reference the field.

field-tag is the field tag, as declared in the SCREEN section.

table is the name or alias of a database table, synonym, or view, as declared in the TABLES section. Qualifiers are not allowed.

Usage

Although you must include an ATTRIBUTES section that assigns at least one name to every *field-tag* from the SCREEN section, you are not required to specify any field attributes.

You are not required to specify *table* unless one of the following is true:

- The *column* identifier occurs in more than one table of the TABLES section.
- *table* is an external table.

If there is ambiguity, **FORM4GL** issues an error during compilation.

Because you can refer to field names collectively through a screen record built upon all the fields linked to the same table, your forms may be easier to work with if you specify *table* for each field. See the INSTRUCTIONS section (page page 5-63) for more information about declaring screen records.

A screen field can display a portion of a character string by using *subscripts* in the *column* specification. Subscripts are a pair of comma-separated integers in square (`[ ]`) brackets to indicate starting and ending character positions within a string value. But if you specify in the ATTRIBUTES section that two fields are linked to the same character column in the database, you *cannot* associate each field with a different substring of the same column.

The ATTRIBUTES section in the following file lists fields linked to columns in the **customer** table. The UPSHIFT and PICTURE attributes that are assigned here are described later in this chapter.

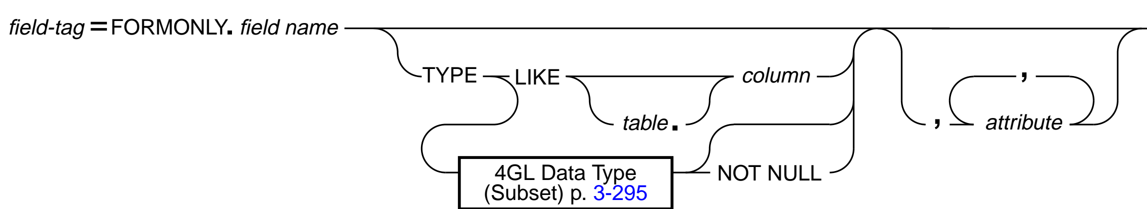
```
DATABASE stores

SCREEN
{
   Customer Name:[f000           ][f001           ]
         Address:[f002                ][f003                 ]
            City:[f004           ] State:[a0] Zip Code:[f005 ]
       Telephone:[f006              ]
}

TABLES   customer

ATTRIBUTES
f000 = customer.fname;
f001 = customer.lname;
f002 = customer.address1;
f003 = customer.address2;
f004 = customer.city;
a0 = customer.state, UPSHIFT;
f005 = customer.zipcode
f006 = customer.phone, PICTURE = "###-###-#### XXXXX";
```

Values from a column of data type BYTE are never displayed in a form; the words `<BYTE value>` are shown in the corresponding display field to indicate that the user cannot see the BYTE data. The following excerpt from a form specification file shows a TEXT field **resume** and a BYTE field **photo**. In this example, the BYTE field is short because only the words `<BYTE value>` are displayed. Similarly, you do not need to include more than one line in a form for a TEXT field. (page 5-50 describes the PROGRAM attribute that can display TEXT or BYTE values.)

```
resume   [f003                                              ]
photo    [f004         ]
   . . .
attributes
f003 = employee.resume
f004 = employee.photo
```

FORMONLY Fields

FORMONLY fields are not associated with columns of any database table or view. They can be used to enter or display the values of program variables. If the DATABASE section specifies FORMONLY, this is the only kind of field description that you can specify in the ATTRIBUTES section.



attribute is a string of keywords, identifiers, and symbols to specify a field attribute, from among those listed on page 5-28.

column is the unqualified SQL identifier of a database column.

field name is an identifier that you assign to a FORMONLY field. This can also appear in 4GL statements that reference the field.

field-tag is a field tag, as declared in the SCREEN section.

table is the name or alias of a table, synonym, or view, as declared in the TABLES section. This is required if columns in different tables of the TABLES section have the same name, or if *table* references an external table or an external, distributed table.

Usage

Like other 4GL identifiers, *field name* cannot begin with a number. It can have up to 50 characters, including letters, numbers, and underscore (_) symbols.

If you specify one or more FORMONLY fields, INFORMIX-4GL behaves as if they formed a database table named **formonly**, with the field names as column names. The following are examples of FORMONLY fields:

```
f021 = FORMONLY.manu_name;
f022 = FORMONLY.unit_price.TYPE.MONEY, COLOR = GREEN;
f023 = FORMONLY.unit_descr.TYPE.LIKE.orders.unit_descr;
f024 = FORMONLY.order_placed
  TYPE DATETIME YEAR TO HOUR NOT NULL, DEFAULT = CURRENT;
```

The Data Type Specification

The optional 4GL data type specification uses a restricted subset of the data type declaration syntax that the DEFINE, ALTER TABLE, or CREATE TABLE statements support. The data type *cannot* be declared here as a RECORD nor as an ARRAY, even if 4GL uses the field to display values from a program record or a program array; *screen arrays* are declared in another section of the form specification file. (It also cannot be SERIAL, since SERIAL is an SQL data type, and only 4GL data types are allowed here.)

If you do not specify any data type, then by default, **FORM4GL** treats the field as type CHAR. Do not assign a length to CHAR, DECIMAL, or MONEY fields, since field length is determined by the display width in the SCREEN section. For example, the demonstration application uses the following FORMONLY field to store the running total price for the order as items are entered:

```
f019 = formonly.t_price;
```

You are required to specify a *data type* only if you also specify an INCLUDE or DEFAULT attribute for this field. 4GL performs any necessary data type conversion for the corresponding program variable during input or display. 4GL evaluates the LIKE clause at compile time, not at run time. If the database schema changes, you may need to recompile a program that uses the LIKE clause to describe a FORMONLY field in a form specification file.

Like a field linked to a database column, a FORMONLY field cannot display a BYTE value directly. The form displays the string `<BYTE value>` to indicate that the user cannot see the BYTE value. Similarly, it is not necessary to allocate more than one line on a form for a FORMONLY field of data type TEXT. You can assign the PROGRAM attribute (page 5-50) to a FORMONLY field to display TEXT or BYTE values from 4GL variables.

The NOT NULL Keywords

The NOT NULL keywords specify that, if you reference this screen field in an INPUT statement, then the user must enter a non-NULL value in the field. (This is more restrictive than the REQUIRED attribute, described on page 5-52, which permits the user to enter a NULL value.)

If the DATABASE section has the WITHOUT NULL INPUT clause (page 5-12), the NOT NULL keyword instructs 4GL to use zero (number or INTERVAL data types) or blanks (character data types) as the default value for this field in INPUT statements. The default DATE value is `12/31/1899`. The default value for DATETIME fields is `1899-12-31 23:59:59.99999`.

Multiple-Segment Fields

If you need to enter or display long character strings from program variables, you can specify *multiple-segment* fields that occupy several lines. To create a multiple-segment field, repeat the same field tag in different fields of the layout in the SCREEN section, typically on successive lines. You must also specify the WORDWRAP attribute for that field tag in the ATTRIBUTES section. During input and display, 4GL treats these as segments of a single field.

The following example shows only the SCREEN and ATTRIBUTES sections of a form specification file that specifies a multiple-segment field:

```
SCREEN SIZE 24 BY 80
{
      title: [title                                   ]
     author: [author                          ]
   synopsis: [synopsis                                          ]
             [synopsis                                          ]
             [synopsis                                          ]
             [synopsis                                          ]
             [synopsis                                          ]
}
      . . .
ATTRIBUTES
    title = booktab.title;
    author = booktab.author;
    synopsis = booktab.synopsis, WORDWRAP COMPRESS;
```

Here the screen field whose tag is **synopsis** appears in five physical segments in the screen layout and has the WORDWRAP attribute. Its value is composed of the physical segments, taken in top-to-bottom, left-to-right order. The field should ordinarily be as long or longer than the program variable or database column that it displays, so it can display all of the text. Users of your 4GL application program may expect all segments to be the same size and laid out in vertical alignment, as in the example, but that is not required. Segments can be of different sizes, and distributed over the screen in any arrangement.

In the description of the field in the last line of the ATTRIBUTES section of the previous example, the keyword WORDWRAP enables a multiple-line editor when the form is open and the cursor enters the field. If you omit it, words cannot flow from segment to segment of the field, and users must move the cursor from field to field with Arrow keys or the ENTER key to edit values in the form. (See the description of the WORDWRAP attribute for more information about the multiple-line editor and about the COMPRESS keyword.)

Field Attributes

Like the ATTRIBUTE clause of various 4GL statements, the ATTRIBUTES section of a form specification file can specify *field attributes.* These optional descriptors can affect the appearance and behavior of individual fields in the screen form. Attributes can specify the display when the cursor is in the field, or can supply or restrict field values during data entry actions:

- Control *cursor movement* among fields (AUTONEXT).
- Set *validation* and *default value* field attributes, including the following:

DEFAULT	NOENTRY	VALIDATE LIKE
INCLUDE	REQUIRED	VERIFY

- Set *formatting* attributes, or automatically invoke a *multiple-line editor* for entry and update of character data, or an *external editor* to view or modify TEXT or BYTE values. The formatting attributes include the following:

COMMENTS	FORMAT	PROGRAM
DISPLAY LIKE	LEFT	UPSHIFT
DOWNSHIFT	PICTURE	WORDWRAP

- Set video display *color* attributes, including the following:

BLACK	GREEN	WHITE
BLUE	MAGENTA	YELLOW
CYAN	RED	

- Set video display *intensity* attributes, including the following:

BOLD	INVISIBLE	REVERSE
DIM	NORMAL	BLINK
UNDERLINE		

A field can have no more than one *color* attribute, and several other attributes are mutually exclusive (DIM and BOLD, UPSHIFT and DOWNSHIFT, NORMAL, and REVERSE). Some field attributes are restricted to certain data types. The DISPLAY LIKE and VALIDATE LIKE attributes have effects that depend on compile-time values in files created by the **upscol** utility of 4GL, as described later in this chapter (page 5-69).

FORM4GL recognizes the following field attributes:

AUTONEXT	FORMAT	REQUIRED
COLOR	INCLUDE	REVERSE
COMMENTS	INVISIBLE	UPSHIFT
DEFAULT	NOENTRY	VALIDATE LIKE
DISPLAY LIKE	PICTURE	VERIFY
DOWNSHIFT	PROGRAM	WORDWRAP

The effects of these field attributes are summarized here:

AUTONEXT	Causes the cursor to advance automatically to the next field.
COLOR	Specifies the color or intensity of values displayed in a field.
COMMENTS	Specifies a message to display on the Comment line.
DEFAULT	Assigns a default value to a field during data entry.
DISPLAY LIKE	Assigns default attributes from table **syscolatt**. The **upscol** utility creates this table, associating default attributes with specific database columns.
DOWNSHIFT	Converts to lowercase any uppercase character data.
FORMAT	Formats DECIMAL, SMALLFLOAT, FLOAT, or DATE output.
INCLUDE	Lists a set of acceptable values during data entry.
INVISIBLE	Does not echo characters on the screen during data entry.
NOENTRY	Prevents the user from entering data into the field.
PICTURE	Imposes a data-entry format on CHAR or VARCHAR fields.
PROGRAM	Invokes an external program to display TEXT or BYTE values.
REQUIRED	Requires the user to supply some value during data entry.
REVERSE	Causes values in the field to be displayed in reverse video.
UPSHIFT	Converts to uppercase any lowercase character data.
VALIDATE LIKE	Validates data entry with values from the **syscolval** table that the **upscol** utility creates, associating default values with specific database columns.
VERIFY	Data must be entered twice when the database is modified.
WORDWRAP	Invokes a multiple-line editor in multiple-segment fields, so that the form can display character strings that are. too long to fit within a single line of the form.

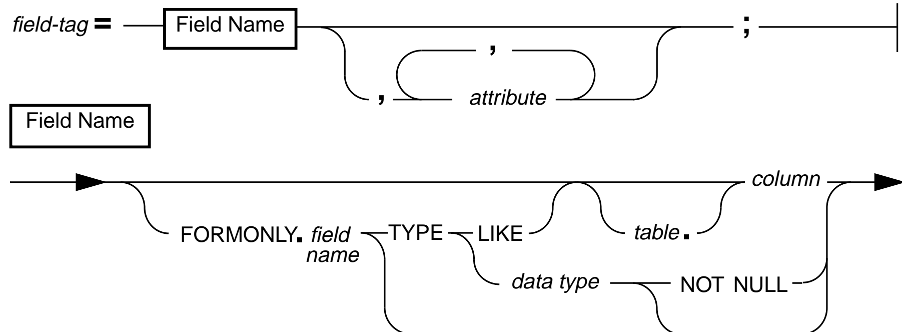
Each of these attributes is described in one of the sections that follow.

Field Attribute Syntax

Syntax for assigning each of these attributes is described in the sections that follow. For simplicity, these use the following syntax diagram format:

```
field-tag = [Field Name] , ——— attribute ———————————————|
```

This format is simplified by ignoring fields with multiple attributes, and by ignoring the required semicolon (;) that separates field descriptions. Here is the complete syntax of a field description in the ATTRIBUTES section:



attribute	is a string of keywords, identifiers, and symbols to specify a field attribute, as described in the sections that follow.
column	is the unqualified SQL identifier of a database column.
data type	is any 4GL data type (page 3-293) except ARRAY or RECORD.
field name	is an identifier that you assign to a FORMONLY field.
field-tag	is the field tag that you declared in the SCREEN section.
table	is the name or alias of a table, synonym, or view, as declared in the TABLES section. This is required if columns in different tables of the TABLES section have the same name, or if *table* references an external table or an external, distributed table.

Note: *If a form links a view to a screen field that permits data entry or data editing, then it is the responsibility of the programmer to test at run time whether the view is updatable, especially if the view is based upon another view.*

AUTONEXT

The AUTONEXT attribute causes the cursor to advance automatically during input to the next field when the current field is full.

```
──► field-tag = [Field Name p. 5-28] , ──── AUTONEXT ──────────────►
```

field-tag is the field tag that you declared in the SCREEN section.

Usage

You specify the order of fields in each INPUT or INPUT ARRAY statement. If the most recent OPTIONS statement specifies INPUT WRAP, the "next" field after the *last* field is the *first* field.

AUTONEXT is particularly useful with character fields in which the input data are of a standard length, such as numeric postal codes, or the abbreviations in the **state** table. It is also useful if a character field has a length of one, since only one keystroke is required to enter data and move to the next field.

If data values entered in the field do not meet requirements of other field attributes like INCLUDE or PICTURE, the cursor does *not* automatically move to the next field, but remains in the current field.

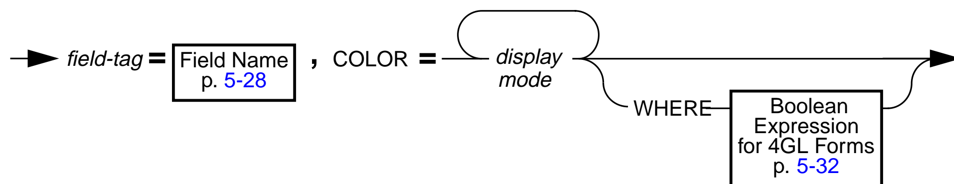
The demonstration application uses the **customer** form to enter all the names and addresses of the customers. The following excerpt from the ATTRIBUTES section of the **customer** form uses the AUTONEXT attribute:

```
a0 = customer.state, DEFAULT = "CA", AUTONEXT;
f007 = customer.zipcode, AUTONEXT;
f008 = customer.phone;
```

When two characters are entered into the **customer.state** field (thus filling the field), the cursor moves automatically to the beginning of the next screen field (the **customer.zipcode** field). When five characters are entered into the **customer.zipcode** field (filling this field), the cursor moves automatically to the beginning of the next field (the **customer.phone** field).

COLOR

The COLOR attribute displays field text in a color or with other video attributes, either unconditionally, or only if a Boolean expression is TRUE.



field-tag is the field tag that you declared in the SCREEN section.

display mode is one of the keywords to specify a *color* or an *intensity*. You can specify zero or one *color*, and zero or more *intensities* from the following lists:

Color Keywords		**Intensity Keywords**
BLACK	MAGENTA	REVERSE
BLUE	RED	LEFT
CYAN	WHITE	BLINK
GREEN	YELLOW	UNDERLINE

Note: *If you are using terminfo, the only color or intensities available are REVERSE and UNDERLINE.*

Usage

If you do not use the WHERE keyword to specify a 4GL Boolean expression, then the intensity and/or color in your *display mode* list applies to the field. This example specifies unconditionally that field text appears in red:

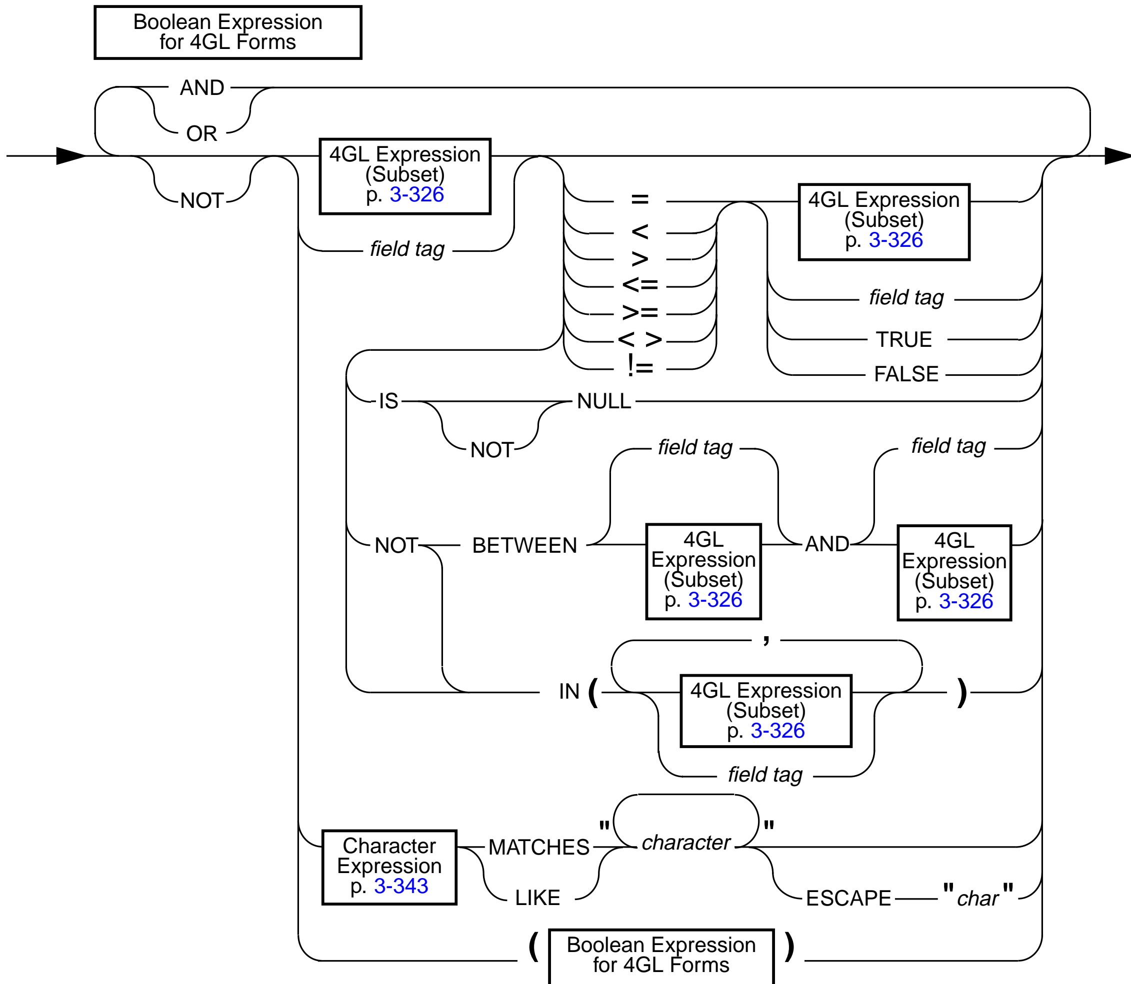
```
f000 = customer.customer_num, COLOR = RED LEFT;
```

Specifying Logical Conditions with the WHERE Option

You can also use the keywords, symbols, and operators that are allowed in 4GL Boolean expressions, including LIKE, MATCHES, TODAY, and CURRENT, in a WHERE clause to specify conditional attributes. If the Boolean expression evaluates as FALSE or NULL, then the field is displayed with default characteristics, rather than with those specified by *display mode*. (See the section "Default Attributes" on page 5-69.)

Boolean Expressions in 4GL Form Specification Files

This is the syntax of 4GL Boolean expressions in the WHERE clause of a COLOR attribute specification.



char is a single ASCII character, enclosed between a pair of single (') or double (") quotation marks.

character is one or more literal or special characters, enclosed between two single (') or double (") quotation marks.

field tag is the field tag (page 5-15) of the current field.

In this diagram, terms for other 4GL *expressions* are restricted subsets. Except for the constants TRUE and FALSE, you cannot reference the *name* of a program variable in the WHERE clause of a COLOR attribute specification. You can, however, include a *field tag* or a *literal* value wherever the name of a variable can appear in a 4GL expression that is a component of the 4GL Boolean expression, as described in the section "Expressions of 4GL" on page 3-326.

If any component of a 4GL Boolean expression is NULL, then the value of the entire 4GL Boolean expression is FALSE (rather than NULL), unless the IS NULL keywords are also included in the expression. Applying the NOT operator to a NULL value does not change its FALSE evaluation. In the following example, the value of the expression is FALSE if a NULL value appears in the display field whose field tag is **f004**:

```
3.1415265 * f004 < 25000
```

If you include a *field tag* in a 4GL Boolean expression when you specify a conditional COLOR attribute, INFORMIX-4GL replaces the *field tag* at run time with the current value in the screen field, and then evaluates the expression.

If *field tag* references a field that is linked to a database column of data type TEXT or BYTE, or to a FORMONLY field of either of those two data types, then only the IS NOT NULL or IS NULL keywords can include that field tag in an expression. The specified color or intensity is applied to the `<TEXT value>` or `<BYTE value>` message, not to the TEXT or BYTE data value, since only the PROGRAM attribute can display a blob value.

Specifying Ranges of Values and Set Membership

A Boolean expression to specify conditional COLOR attributes can include SQL Boolean operators that are not valid in ordinary 4GL Boolean expressions. You can use the BETWEEN ... AND operator to specify a range of number, time, or character values. Here the first expression cannot be *greater* than the second (for number expressions), nor *later* than the second (for time expressions), nor *later in the ASCII collating sequence* than the second (for character expressions). Appendix A lists the numeric values of ASCII characters.

The WHERE clause of a COLOR field description can also use the IN operator to specify a comma-separated list (enclosed between parentheses) of values with which to compare the field tag or expression.

(Chapter 3 describes the syntax of the BETWEEN ... AND and IN Boolean operators in conditional COLOR specifications for 4GL forms. See also the *Informix Guide to SQL: Reference* for the complete syntax of Boolean expressions in SQL statements.)

Data Type Compatibility

You may get unexpected results if you use relational operators or the BETWEEN, AND, or IN operators with expressions of dissimilar data types. In general, you can compare numbers with numbers, character strings with character strings, and time values with time values.

If a *time expression* component of a 4GL Boolean expression is an INTERVAL data type, then any other time expression that is compared to it by a relational operator must also be an INTERVAL value. You cannot compare a span of time (an INTERVAL value) with a point in time (a DATE or DATETIME value).

Data-Type Conversion in 4GL Boolean Expressions

If you specify a number, character, or time expression in a context where a 4GL Boolean expression is expected, INFORMIX-4GL applies the following rules after evaluating the number, character, or time expression:

- If the value is a non-zero real number (or a character string representing a non-zero number) or a non-zero INTERVAL, or any DATE or DATETIME value, then the 4GL Boolean value is TRUE.
- If the value is NULL, and the IS NULL keywords are also included in the expression, then the value of the 4GL Boolean expression is TRUE.
- Otherwise, the 4GL Boolean expression is FALSE.

The Display Modes

The display and intensity keywords of the COLOR attribute have the same effects on a field as the same keywords of the **upscol** utility (page 5-69) or ATTRIBUTES clause (page 3-290). The following table shows the effects of the color attribute keywords on a monochrome terminal, as well as the effects of the intensity attribute keywords on a color terminal:

Attribute	Display	Attribute	Display
WHITE	normal	NORMAL	white
YELLOW	bold	BOLD	red
MAGENTA	bold	DIM	blue
RED	bold		
CYAN	dim		
GREEN	dim		
BLUE	dim		
BLACK	dim		

The LEFT attribute produces a left-justified display in a screen field of any *number* data type. It has no effect on fields of other data types. (Without the COLOR = LEFT specification, number values are right-justified by default.)

The next lines specify display attributes if Boolean expressions are TRUE:

```
f001  = FORMONLY.late, COLOR = RED BLINK WHERE f001 < TODAY;
f002  = manufact.manu_code, COLOR = RED WHERE f002 = "HRO";
f003  = customer.lname, COLOR = RED WHERE f003 LIKE "Quinn";
f004  = mytab.col6, COLOR = GREEN WHERE f004 < 10000;
f005  = mytab.col9, COLOR = BLUE REVERSE WHERE f005 IS NULL,
        COLOR = YELLOW WHERE f005 BETWEEN 5000 AND 10000,
        COLOR = GREEN BLINK WHERE f005 > 10000;
```

The following expression is TRUE if the field **f022** does not include the underscore character:

```
NOT f022 LIKE "%z_%" ESCAPE "z"
```

Related Attributes

DISPLAY LIKE, INVISIBLE, REVERSE

COMMENTS

The COMMENTS attribute displays a message on the Comment line at the bottom of the window. The message is displayed when the cursor moves to the specified field, and is erased when the cursor moves to another field.

field-tag = [Field Name p. 5-28] , COMMENTS = "*message*"

field-tag is the field tag that you declared in the SCREEN section.

message is a character string enclosed in quotation marks.

Usage

The *message* must appear between quotation (") marks on a single line of the form specification file.

In the following example, the field description specifies a *message* for the Comment line to display. The message will appear when the screen cursor enters the field that is linked to the **fname** column of the customer table. In the **stores** database, this column contains the first name of a customer:

```
c2 = customer.fname, comments =
   "Please enter initial if available.";
```

The most common application of the COMMENTS attribute is to give information or instructions to the user. This is particularly appropriate when the field accepts only a limited set of values. (See the description of the INCLUDE attribute later in this section for details of how to specify a range or a list of acceptable values for data entry.)

4GL programs can use the same screen form to support several distinct tasks (for example, data input and query by example). Do not specify the COMMENTS attribute in a field description unless the *message* is appropriate to all of the tasks in which the *message* can appear.

If the same field requires a different message for various tasks, you should specify each message using the MESSAGE or DISPLAY statements, rather than in the form specification file.

The Position of the Comment Line

The default position of the Comment line in the 4GL screen is line 23. You can reset this position with the OPTIONS statement.

The default position of the Comment line in a 4GL window is LAST. You can reset this position in the OPTIONS statement, if you want the new position in all 4GL windows. Alternatively, you can reset it in the ATTRIBUTE clause of the appropriate OPEN WINDOW statement, if you want the new position in a specific 4GL window. Chapter 3 describes the OPTIONS and OPEN WINDOW statements.

Related Attribute

INCLUDE

DEFAULT

The DEFAULT attribute assigns a default value to a field during data entry.

field-tag = [Field Name p. 5-28] , DEFAULT = *value*

field-tag is the field tag that you declared in the SCREEN section.

value is a default value for the field. This restricted expression cannot reference any variable nor programmer-defined function.

Usage

Default values have no effect when you execute the INPUT statement using the WITHOUT DEFAULTS option. In this case, 4GL displays the values in the program variables list on the screen. The situation is the same for the INPUT ARRAY statement, except that 4GL displays the default values when the user inserts a new row.

If the field is FORMONLY, you must also specify a data type when you assign the DEFAULT attribute to a field. (See "FORMONLY Fields" on page 5-24.)

If both the DEFAULT attribute and the REQUIRED attribute are assigned to the same field, then the REQUIRED attribute is ignored.

If you do not use the WITHOUT NULL INPUT option in the DATABASE section, all fields default to NULL values unless you use the DEFAULT attribute. If you use the WITHOUT NULL INPUT option in the DATABASE section and you do not use the DEFAULT attribute, then character fields default to blanks, number and INTERVAL fields to `0`, and MONEY fields to `$0.00`. The default DATE value is `12/31/1899`. The default DATETIME value is `1899-12-31 23:59:59.99999`.

You cannot assign the DEFAULT attribute to fields of data type TEXT or BYTE.

Literal Values

The *value* can be a quoted string, a literal number (page 3-342), a literal DATE value (page 3-349), a literal DATETIME value (page 3-351), a literal INTERVAL value (page 3-355), or a built-in function or operator (page 4-11) that returns a single value of a data type compatible with that of the field.

If you include in the *value* list a character string that contains a blank space, a comma(,) symbol, or any special characters, or a string that does not begin with a letter, then you must enclose the entire string in quotation (") marks.

(If you omit the quotation marks, any uppercase letters are downshifted.)

For a DATE field, you must enclose any literal *value* in quotes ("). For a DATETIME or INTERVAL, field, you can enclose *value* in quotation (") marks, or you can enter it as an unquoted literal:

DATETIME (*data-value*) *qualifier*
INTERVAL (*data-value*) *qualifier*
– INTERVAL (*data-value*) *qualifier*

Pages 3-349 through 3-355 describe DATETIME and INTERVAL literals.

Built-In 4GL Operators and Functions as Values

Besides these literal values, you can also specify a built-in **4GL** function or operator that returns a single value of the appropriate data type. Arguments or operands must be a literal value, a built-in **4GL** function or operator that returns a single value, or the named constants TRUE or FALSE. For example, a default value of data type INTERVAL can be specified in the format:

integer UNITS *time-unit*

Here *integer* can be a positive or negative literal integer (page 3-340), or an expression in parentheses that evaluates to an integer, and *time-unit* is a keyword from an INTERVAL qualifier, such as MONTH, DAY, HOUR, and so forth. (This must be consistent with the explicit or implied data type declaration of the field; do not, for example, specify YEAR or MONTH as the *time-unit* for a DAY TO FRACTION field.)

Use the TODAY operator as the *value* to assign the current date as the default value of a DATE field. Use the CURRENT operator as the *value* to assign the current date and time as the default for a DATETIME field. (**4GL** does not assign these values automatically as defaults, so you must specify them explicitly.) These are evaluated at run time, not at compile time.

The following field descriptions specify DEFAULT values:

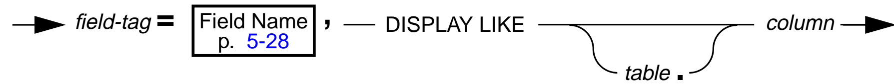
```
c8 = state, UPSHIFT, AUTONEXT,
    DEFAULT = "CA";
o12 = order_date, DEFAULT = TODAY;
f019 = FORMONLY.timestamp TYPE DATETIME YEAR TO DAY
    COLOR = RED, DEFAULT = CURRENT;
```

Related Attributes

INCLUDE, REQUIRED, VALIDATE LIKE

DISPLAY LIKE

The DISPLAY LIKE attribute applies the attributes that the **upscol** utility assigned to a specified column in the **syscolatt** table to a field.



field-tag is the field tag that you declared in the SCREEN section.

table is the unqualified name or the alias of a database table, synonym, or view, as declared in the TABLE section. (This is not required unless several columns in different tables have the same name, or if the table is an external table or an external, distributed table.)

column is the name of a column in *table*, or (if you omit *table*) the unique identifier of a column in one of the tables that you declared in the TABLES section.

Usage

Specifying this attribute is equivalent to listing all the attributes that are assigned to *table.column* in the **syscolatt** table. (The section "Default Attributes" on page 5-69 describes the **syscolatt** table. See also the description of the **upscol** utility in Appendix B.)

You do not need to specify the DISPLAY LIKE attribute if the field is linked to *table.column* in the Field Name specification.

You cannot specify a column of data type BYTE as *table.column*.

The following example instructs INFORMIX-4GL to apply the default display attributes of the **items.total_price** column to a FORMONLY field.

```
s12 = FORMONLY.total, DISPLAY LIKE items.total_price;
```

4GL evaluates the LIKE clause at compile time, not at run time. If the database schema changes, you may need to recompile a program that uses the LIKE clause. Even if all of the fields in the form are FORMONLY, this attribute requires **FORM4GL** to access the database that contains *table*.

Related Attribute

VALIDATE LIKE

DOWNSHIFT

Assign the DOWNSHIFT attribute to a character field when you want INFORMIX-4GL to convert uppercase letters entered by the user to lowercase letters, both on the screen and in the corresponding program variable.

```
──► field-tag = [Field Name p. 5-28] , ── DOWN SHIFT ──────────►
```

field-tag is the field tag that you declared in the SCREEN section.

Usage

Because uppercase and lowercase letters have different ASCII values, storing character strings in one or the other format can simplify sorting and querying a database.

By specifying the DOWNSHIFT attribute, you instruct INFORMIX-4GL to convert character input data to lowercase letters in the program variable.

The maximum length of a character value to which you can apply the DOWNSHIFT attribute is 511 characters.

NLS

When NLS is active, the results of conversion between uppercase and lowercase are appropriate to the national language in use, as defined by the LC_CTYPE environment variable.

Related Attribute

UPSHIFT

FORMAT

You can use the FORMAT attribute with a DECIMAL, SMALLFLOAT, FLOAT, or DATE field to control the format of output displays.

```
——► field-tag = [Field Name p. 5-28] , FORMAT = "format-string" ——►
```

field-tag is the field tag that you declared in the SCREEN section.

format-string is a string of characters to specify a data display format. You must enclose *format-string* within quotation (") marks.

Usage

This attribute can format data that the application displays in the field. (Use the PICTURE attribute to format data that are entered in the field by the user.) INFORMIX-4GL displays the data right-justified in the field.

If the *format-string* is smaller than the field width, **FORM4GL** issues a compile-time warning, but the form is usable.

Formatting DATE Values

For DATE data types, INFORMIX-4GL recognizes the following symbols as special in the *format-string*:

mm produces the two-digit representation of the month; for example, `Jan` = `01`, `Feb` = `02`, and so on.

mmm produces a three-letter English language abbreviation of the month; for example, `Jan`, `Feb`, and so on.

dd produces the two-digit representation of the day of the month.

ddd produces a three-letter English language abbreviation of the day of the week; for example, `Mon`, `Tue`, and so on.

yy produces the two-digit representation of the year, discarding the leading digits. The year 2003, for example, would appear as `03`.

yyyy produces a four-digit representation of the year.

For DATE fields, **FORM4GL** interprets any other characters as literals, and displays them wherever you place them within *format-string*.

Here are some example format strings and their corresponding display formats for DATE fields that display the 15th day of September, 1993:

Input	Result
no FORMAT *attribute*	09/15/1993
FORMAT = “mm/dd/yy”	09/15/93
FORMAT = “mmm dd, yyyy”	Sep 15, 1993
FORMAT = “yymmdd”	930915
FORMAT = “dd-mm-yy”	15-09-93
FORMAT = “(ddd.) mmm. dd, yyyy”	(Wed.) Sep. 15, 1993

Formatting Number Values

For DECIMAL, SMALLFLOAT, or FLOAT data types, the *format-string* consists of pound signs (#) that represent digits, and a decimal point. For example, “###.##” produces at least three places to the left of the decimal point and exactly two to the right.

If the actual number displayed requires fewer characters than the *format-string*, 4GL right-justifies it, and pads the left with blanks.

If necessary to satisfy the *format string*, 4GL rounds number values before it displays them.

NLS

When NLS is active, the setting in the NLS environment variable LC_NUMERIC affects the way the format string in the FORMAT attribute is interpreted for numeric data. In the format string, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma symbol (,) is a placeholder for the thousands separator specified by environment variables. Thus, the format string #,###.## formats the value 1234.56 as 1,234.56 in a US English locale but as 1.234,56 in a German locale.

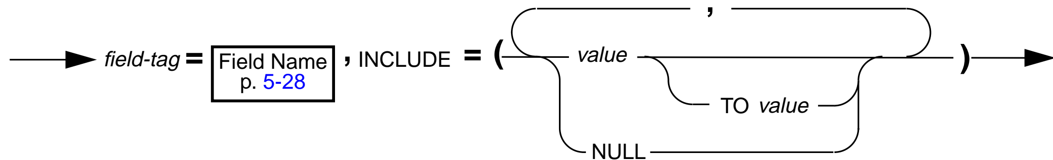
The *mmm* and *ddd* specifiers in a format string can display language-specific month name and day name abbreviations. This requires the installation of message files in a subdirectory of **$INFORMIXDIR/msg**, and subsequent reference to that subdirectory by way of the environment variable DBLANG. For example, in a Spanish locale, the *ddd* specifier translates the day Saturday into the day name abbreviation Sab, which stands for Sabado (the Spanish word for Saturday). For more information on NLS, see Appendix E.

Related Attribute

PICTURE

INCLUDE

The INCLUDE attribute specifies acceptable values for a field, and causes 4GL to check at run time before accepting an input value.



field-tag is the field tag that you declared in the SCREEN section.

value is an element in a comma-separated list (within parentheses) of values (*value1*, *value2*, . . .), or a range of values (*value1* TO *value2*), or any combination of individual values and ranges.

Usage

Each *value* is a restricted expression that cannot include the name of any **4GL** variable nor programmer-defined function. It can include literal values, built-in functions and operators (page 4-11), and the constants TRUE and FALSE. The same rules for DEFAULT attribute values (page 5-38) also apply to INCLUDE values. TEXT fields and BYTE fields cannot have this attribute.

If a field has the INCLUDE attribute, the user must enter an acceptable value (from the *value* list) before INFORMIX-4GL accepts a new row. If the *value* list does not include the default value, then the INCLUDE attribute behaves like the REQUIRED attribute, and an acceptable entry is required. Include the NULL keyword in the *value* list to specify that it is acceptable for the user to press the ENTER key without entering any value.

```
f006 = survey.item06, INCLUDE = (NULL, "YES", "NO");
```

In this example, the NULL keyword allows the user to enter nothing. You *cannot* specify this by substituting a string of blanks for the NULL keyword here, since a NULL character value is different from ASCII 32, the blank character.

Including a COMMENTS attribute to describe acceptable values makes data entry easier, since you can display a message to advise the user of whatever restrictions you have imposed on data entry:

```
i18 = items.quantity, INCLUDE = (1 TO 50),
    COMMENTS = "Acceptable values are 1 through 50";
```

If you include in the *value* list a character string that contains a blank space, a comma (,) symbol, or any special characters, or a string that does not begin with a letter, then you must enclose the entire string in quotation (") marks. (If you omit the quotation marks, any uppercase letters are downshifted.)

Ranges of Values

You can use the TO keyword to specify an inclusive range of acceptable values. For example, ranges in the following field description include the postal abbreviations for the names of the contiguous states of the United States:

```
i20 = customer.state,
INCLUDE = (NULL, "AL" TO "GA", "IA" TO "WY"),
COMMENTS = "No Alaska (AK) or Hawaii (HI) addresses here.";
```

When you specify a range of values, the *lower* value must appear first. The meaning of "lower" depends on the data type of the field:

- For number or INTERVAL fields, this is the larger (or only) negative value, or (if neither value is negative) the value closer to zero.
- For other time fields, it is the earlier DATE or DATETIME value.
- For character fields, the "lower" value is the string that starts with a character closer to the beginning of the ASCII collating sequence. (See Appendix G for a listing of the ASCII collating sequence.)

In a number field, for example, the range "`5 TO 10`" is valid. In a character field, however, it produces a compile-time error. (The character string "`10`" is less than "`5`" because 1 comes before 5 in the ASCII collating sequence.)

FORMONLY Fields

You must specify a data type when you assign the INCLUDE attribute to a FORMONLY field (page 5-24). The TYPE clause is required in the following example, rather than optional:

```
    f006 = FORMONLY.item07 TYPE CHAR(*),
         INCLUDE = (NULL, "PERHAPS", "MAYBE");
```

Related Attributes

COMMENTS, DEFAULT, REQUIRED

INVISIBLE

The INVISIBLE attribute prevents user-entered data from being echoed on the screen during CONSTRUCT, INPUT, INPUT ARRAY, or PROMPT statement.

```
──────► field-tag = [Field Name p. 5-28] , ──── INVISIBLE ──────►
```

field-tag is the field tag that you declared in the SCREEN section.

Usage

Characters that the user enters in a field with this attribute are not displayed during data entry, but the cursor moves through the field as the user types. No other aspects of data entry are affected by the INVISIBLE attribute.

The following example illustrates the use of the INVISIBLE attribute:

```
i001 = FORMONLY.secret_password TYPE LIKE state.sname,
    INVISIBLE,
    COMMENTS = "Enter your secret password.";
```

If you specify INVISIBLE and any other display attribute for a field, then 4GL ignores the INVISIBLE attribute.

The INVISIBLE attribute has no effect on editing BYTE or TEXT fields.

This attribute does *not* prevent a DISPLAY, DISPLAY ARRAY, DISPLAY FORM, MESSAGE, or OPEN WINDOW statement from displaying data in the field.

Specify the INVISIBLE attribute, rather than COLOR = BLACK, if you do not want the field to display what the user types during data entry. (The BLACK color attribute displays black characters on a color or monochrome terminal.)

Related Attribute

COLOR

NOENTRY

The NOENTRY attribute prevents data entry into the field during an INPUT or INPUT ARRAY statement.

```
──► field-tag = [Field Name p. 5-28] , ─── NOENTRY ───►
```

field-tag is the field tag that you declared in the SCREEN section.

Usage

The following example illustrates the use of the NOENTRY attribute:

```
i13 = stock.stock_num, NOENTRY;
```

When the user enters data into the **stock** table, the **stock_num** column is not available, since this SERIAL column gets its value from the database engine during the INSERT statement.

The NOENTRY attribute does *not* prevent data entry into a field during a CONSTRUCT statement (for a query by example).

Related Attribute

INVISIBLE

PICTURE

The PICTURE attribute specifies a character pattern for data entry into a text field, and prevents entry of values that conflict with the specified pattern.

→ *field-tag* = [Field Name p. 5-28] , PICTURE = "*format-string*" →

field-tag is the field tag that you declared in the SCREEN section.

format-string is a string of characters to specify a character pattern for data entry. This must be enclosed within quotation (") marks.

Usage

A *format-string* can include literals and these three special symbols:

Symbol	Meaning
A	Any letter
#	Any digit
X	Any character

4GL treats any other character in the *format-string* as a literal. The cursor skips over any literals during data entry. 4GL displays the literal characters in the display field and leaves blanks elsewhere.

For example, the field specification

```
c10 = customer.phone,
     picture = "###-###-####x#####";
```

displays these symbols in the **customer.phone** field before data entry:

```
[   -   -    x     ]
```

If the user attempts to enter a character that conflicts with the *format-string*, then the terminal beeps, and 4GL does not echo the character on the screen.

The *format-string* must fill the entire width of the display field. The PICTURE attribute, however, does not require data entry into the entire field. It only requires that whatever characters are entered conform to *format-string*.

When PICTURE specifies input formats for DATETIME or INTERVAL fields, **FORM4GL** does not check the syntax of *format-string*, but your form will work if the syntax is correct. Any error in *format-string*, however, such as an incorrect field separator, produces a run-time error.

As another example, if you specify a field for part numbers like this

```
f1 = part_no, picture = "AA#####-AA(X)";
```

then INFORMIX-4GL accepts any of the following inputs:

```
LF49367-BB(*)
TG38524-AS(3)
YG67489-ZZ(D)
```

The user does not enter the "-" nor the parentheses, but INFORMIX-4GL includes them in the string that it passes to the program variable.

NLS

The PICTURE attribute is not affected by NLS environment variables because it only formats character information.

Editing Keys During Data Entry

INFORMIX-4GL supports CONTROL-D and CONTROL-X in fields that specify the PICTURE attribute:

- CONTROL-D deletes characters from the current cursor position to the end of the field.
- CONTROL-X deletes the current character.

Related Attribute

FORMAT

PROGRAM

The PROGRAM attribute can specify an external application program to work with screen fields of data type TEXT or BYTE.

field-tag = [Field Name p. 5-28] , PROGRAM = "*command*"

command is a command string (or the name of a shell script) that invokes an editing program, enclosed within quotes.

field-tag is the field tag that you declared in the SCREEN section.

Usage

You can assign the PROGRAM attribute to a BYTE or TEXT field to call an external program to work with the TEXT or BYTE values. Users of the application invoke the external program by pressing the exclamation (**!**) point key while the screen cursor is in a blob field. The external program then takes over control of the screen. When the user exits from the external program, the form is redisplayed, with any display attributes besides PROGRAM in effect.

For example, this field description designates **vi** as the external editor of a multiple-segment TEXT field that also has the WORDWRAP attribute:

```
f010 = personnel.resume, WORDWRAP, PROGRAM = "vi";
```

Here the WORDWRAP attribute (described on page 5-57) specifies that as much of the TEXT value as possible be displayed in successive segments of the multiple-segment field when the form is displayed with a value in the field; the WORDWRAP editor *cannot* be used to edit a TEXT value.

If the user moves the cursor into the field whose tag is `f010` in the same example and presses the `!` key, the form is cleared from the screen. Now the user can run the **vi** utility to create, examine, or modify the TEXT value. When the **vi** editing session ends, the screen form is restored on the screen, and control returns to the *4GL* application.

When a display field is of data type TEXT, but the screen cursor is not in that field, the INFORMIX-4GL application program can display as many of the leading characters of a TEXT data value as can fit in the screen field. (If the field is of data type BYTE, then INFORMIX-4GL displays `<BYTE value>` in the field.) This behavior is independent of the PROGRAM attribute.

Default Editors

If a user moves the cursor into a TEXT field and presses the exclamation (!) point key in the first character position of the field, INFORMIX-4GL attempts to invoke an external program. The program invoked for a TEXT field is chosen from among the following, in *descending* order of priority:

- The program (if any) identified by the PROGRAM ="command" attribute specification for the field.
- The program (if any) named in the DBEDIT environment variable.
- The default editor, which depends on the host operating system.

Specify the editor to use with the DBEDIT environment variable; this variable should contain the name of a UNIX application such as **vi** or **emacs**. When the user exits the editor, control returns to the 4GL screen.

4GL applications that display or modify a value in a BYTE field must use the PROGRAM attribute explicitly to assign an editor. For BYTE fields, the default editor is not called, and the DBEDIT variable is not examined.

The Command String

Before invoking the program, your application copies the BYTE or TEXT field to a temporary disk file. It then issues a system command composed of the *command* that you specify after the PROGRAM keyword, followed by the name of the temporary file.

The *command* string can be longer than a single word. You can add additional command parameters. The *command* can also be the name of a shell script, so that you can initiate a whole series of actions.

Your *4GL* program needs to execute an INSERT or UPDATE statement using the appropriate program variables after input is terminated. For example, you would use a statement like one of the following:

```
INSERT INTO mytable (textcol, bytecol)
   VALUES (p_texdata, p_bytdata)

UPDATE mytable SET (textcol, bytecol) = (p_texdata, p_bytdata)
```

REQUIRED

The REQUIRED attribute forces the user to enter data into the field during an INPUT or INPUT ARRAY statement.

field-tag = [Field Name p. 5-28] , —— REQUIRED ——►

field-tag is the field tag that you declared in the SCREEN section.

Usage

The REQUIRED keyword is effective only when the *field name* occurs in the list of screen fields of an INPUT or INPUT ARRAY statement. For example, suppose the ATTRIBUTES section includes the following field description:

```
o20 = orders.po_num, REQUIRED;
```

Because of the REQUIRED specification, 4GL requires the entry of a purchase order value when the form is used to collect information for a new order.

You cannot specify a default value for a REQUIRED field. If both the REQUIRED and the DEFAULT attributes are assigned to the same field, then 4GL assumes that the DEFAULT value satisfies the REQUIRED attribute.

This attribute requires only that the user enter a printable character in the field. If the user subsequently erases the entry during the same input, 4GL considers the REQUIRED attribute satisfied. If you want to insist on a non-NULL entry, specify that the field is FORMONLY and NOT NULL.

Related Attribute

NOENTRY

REVERSE

The REVERSE attribute displays any value in the field in reverse video (dark characters in a bright field).

field-tag = Field Name p. 5-28 , REVERSE

field-tag is the field tag that you declared in the SCREEN section.

Usage

The following example specifies that a field linked to the **customer_num** column displays data in reverse (sometimes called “inverse”) video:

```
f000 = customer.customer_num, REVERSE;
```

On terminals that do not support reverse video, fields having the REVERSE attribute are enclosed between angle brackets (< >) symbols.

The REVERSE attribute disables any other COLOR attribute for the same field.

Related Attribute

COLOR

UPSHIFT

During data entry in a character field, the UPSHIFT attribute converts lowercase letters to uppercase letters, both on the screen display, and in the 4GL program variable that stores the contents of that field.

field-tag = [Field Name p. 5-28] , UPSHIFT

field-tag is the field tag that you declared in the SCREEN section.

Usage

Because uppercase and lowercase letters have different ASCII values, storing all character strings in one or the other format can simplify sorting and querying a database.

The following example includes UPSHIFT in the attribute list of a field:

```
c8   = state, UPSHIFT, AUTONEXT,
       INCLUDE = ("CA", "OR", "NV", "WA"),
       DEFAULT = "CA" ;
```

Because of the UPSHIFT attribute, INFORMIX-4GL enters uppercase characters in the **state** field regardless of the case used to enter them.

The AUTONEXT attribute tells INFORMIX-4GL to move automatically to the next field once you type the total number of characters allowed for the field (in this instance, two characters). The INCLUDE attribute restricts entry in this field to the characters CA, OR, NV, or WA only. The DEFAULT value for the field is CA.

NLS

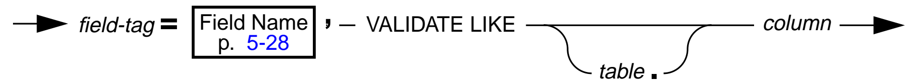
When NLS is active, the results of conversion between uppercase and lowercase are appropriate to the national language in use, as defined by the LC_CTYPE environment variable.

Related Attribute

DOWNSHIFT

VALIDATE LIKE

The VALIDATE LIKE attribute instructs 4GL to validate the data entered into the field, using the validation rules that the **upscol** utility assigned to the specified database column in the **syscolval** table.



column is the name of a column in *table*, or (if you omit *table*) the unique identifier of a column in one of the tables that you declared in the TABLES section.

field-tag is the field tag that you declared in the SCREEN section.

table is the unqualified name or the alias of a database table, synonym, or view, as declared in the TABLE section. (This is not required unless several columns in different tables have the same name, or if the table is an external table or an external, distributed table.)

Usage

This attribute is equivalent to listing all the attributes that you have assigned to *table.column* in the **syscolval** table. "Default Attributes" on page 5-69 describes the **syscolval** table, and the effects of this table in an ANSI-compliant database. The following example assigns the default attributes of the **customer.state** column to a FORMONLY field:

```
s13 = FORMONLY.state, VALIDATE LIKE customer.state;
```

The restrictions on the DISPLAY LIKE attribute also apply to this attribute. You do not need the VALIDATE LIKE attribute if *table.column* is the same as *field name*. You cannot specify a column of data type BYTE as *table.column*.

Even if all of the fields in the form are FORMONLY, this attribute requires **FORM4GL** to access the database that contains *table*.

Related Attribute

DISPLAY LIKE

VERIFY

The VERIFY attribute requires users to enter data into the field twice, in order to reduce the probability of erroneous data entry.

```
──► field-tag = [Field Name p. 5-28] , ──── VERIFY ──────────────►
```

field-tag is the field tag that you declared in the SCREEN section.

Usage

Since some data are critical, this attribute supplies an additional step in data entry to ensure the integrity of your data. After the user enters a value into a VERIFY field and presses ENTER, 4GL erases the field and requests reentry of the value. The user must enter *exactly* the same data each time, character for character: `15000` is not exactly the same as `15000.00`.

For example, if you specify a field for salary information in this way

```
s10 = quantity, VERIFY;
```

then 4GL requires entry of exactly the same data twice. An error message appears if the user does not enter the same keystrokes.

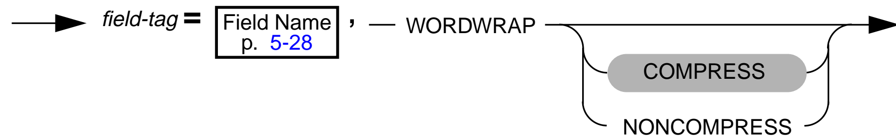
The VERIFY attribute takes effect while INPUT, INPUT ARRAY, or UPDATE statements of *4GL* are executing. It has no effect on CONSTRUCT statements.

Related Attributes

INCLUDE, REQUIRED, VALIDATE LIKE

WORDWRAP

In a multiple-segment field, the WORDWRAP attribute enables a multiple-line editor. This can "wrap" long character strings to the next line of a multiple-segment field for data entry, data editing, and data display.



field-tag is the field tag that you declared in the SCREEN section, where it appears in two or more field segments.

Usage

If the same field tag is repeated in two or more locations in the screen layout, this attribute instructs INFORMIX-4GL to treat all the instances of that field tag as successive segments of a *multiple-segment field* (page 5-26). These can display data strings that are too long to fit on a single line of the screen form. For example, the following excerpt from a form specification file shows a VARCHAR field linked to the **history** column in the **employee** table.

```
history      [f002                                            ]
             [f002                                            ]
             [f002                                            ]

attributes
f002 = employee.history, WORDWRAP COMPRESS;
```

4GL replaces each set of multiple-segment fields with a single WORDWRAP field of a rectangular shape. The COMPRESS keyword option is applied to this field, and the delimiters are replaced with blank spaces.

A multiple-segment field can also have an irregular shape like the following:

```
[TightFit #34B, "The Big Squeeze,"  ]
[is the most amazing compression    ]
[utility ever written! Use this new ]
 Copyright 1992 [version for any    ]
 TightFit Corp. [application that   ]
 54678 Squeeze Blvd.[requires truly ]
 Ste. 456XZ [efficient use of space ]
 Nowheresville [at rock-bottom      ]
 MN, 23456-6789 [cost. Our new site ]
 Tel.#:(925) [licenses allow        ]
 123-4567 Ext. 34 [multiple users to]
```

If the lines of the multiple-segment field are not contiguous, or if the field has an irregular shape, as in the previous example, the WORDWRAP field that results is based on the maximum height and width of the multiple-segment field as a unit. The resulting WORDWRAP field can overlap or be overlapped by labels or individual form fields. To prevent such unpredictable effects, consolidate the segments of multiple-segment fields into rectangular shapes.

When a variable is bound to the WORDWRAP field during INPUT, only the number of characters allowed by the bound variable can be entered. If necessary, text in the field scrolls to allow the full number of characters to be entered. Data compression takes place before storage in the bound variable.

Data Entry and Editing with WORDWRAP

When text is entered into a multiple-segment field whose attributes include WORDWRAP, INFORMIX-4GL breaks character strings into segments at blanks (if it can), padding field segments with blanks at the right. Where possible, contiguous non-blank substrings (here called "words") within a string are not broken at field segment boundaries.

When keyboard input reaches the end of a line, the multiple-line editor brings the current word down to the next field segment, moving text down to subsequent lines as necessary. (The "next" field segment is determined by the left-to-right, top-to-bottom order of field segments within the screen layout.) When the user deletes text, the editor pulls words up from lower field segments whenever it can.

Data Display with WORDWRAP

If a CHAR, VARCHAR, or TEXT value is displayed in a multiple-segment field, INFORMIX-4GL displays the first data character in the first character position of the first segment, and displays consecutive data characters in successive positions to the right. If the entire data string is too long to fit in the first field

segment, INFORMIX-4GL continues the display in the next field segment, dividing the data string at blank characters. This process continues until all the field segments are filled, or until the end of the data string is reached.

The WORDWRAP attribute displays a TEXT field so that it fits into the form without any field segments beginning with a blank. For a TEXT field, the WORDWRAP attribute only affects how the value is displayed; WORDWRAP does not enable the multiple-line editor. To let users edit a TEXT field, you must use the PROGRAM attribute to indicate the name of an external editor.

Displaying Program Variables with WORDWRAP

Text in WORDWRAP fields can include printable ASCII characters, the TAB (ASCII 9) character, and the NEWLINE (ASCII 10) character. These are retained in the program variable. Other non-printable characters may result in run-time errors. The TAB character aligns the display at the next tab stop, while NEWLINE continues the display at the start of the next line. By default, tab stops are in every eighth column, beginning at the left-hand edge of the field.

Ordinarily, the length of the variable should not be greater than the total length of all the field segments. If data are longer than the field (or if too much padding is required for WORDWRAP), 4GL fills the field and discards the excess data. This displays a long variable in summary form. If a truncated variable is used to update the database, however, characters are lost.

The editor distinguishes between *intentional* blanks (from the database or typed by the user) and *editor* blanks (inserted at the ends of lines for word-wrap or to align after a NEWLINE). Intentional blanks are retained as part of the data. Editor blanks are inserted and deleted automatically as required.

When designing a multiple-segment field, you should allow room for editor blanks, over and above the data length. The expected number of editor blanks is half the length of an average word per segment. Text that requires more space than you expect might be truncated after the final field segment.

The COMPRESS Option of WORDWRAP

The COMPRESS keyword prevents blanks produced by the editor from being included in the program variable. COMPRESS is applied by default and causes truncation to occur if the sum of intentional characters exceeds the field or column size. Because of editing blanks in the WORDWRAP field, the stored value may not correspond exactly to its multiple-line display, so a 4GL report generally cannot display the data in identical form.

In the following fragment of a form specification file, a CHAR value in the column **charcolm** is displayed in the multiple-segment field whose tag is **mlf**.

```
SC0REEN SIZE 24 by 80
{
Enter text:
    [mlf                                                ]
    [mlf                                                ]
         . . .
    [mlf                                                ]
    [mlf                                                ]
}

TABLES  tablet . . .

ATTRIBUTES
    mlf = tablet.charcolm, WORDWRAP COMPRESS;
```

If the data string is too long to fit in the first line, successive segments are displayed in successive lines, until all of the lines are filled, or until the last text character is displayed (whichever happens first).

If the form is used to insert data into **tablet.charcolm**, the keyword COMPRESS specifies that INFORMIX-4GL will not store editor blanks.

WORDWRAP Editing Keys

When data are entered or updated in a WORDWRAP field, the user can use keys to move the screen cursor over the data, and to insert, delete, and type over the data. The cursor never pauses on editor blanks.

The editor has two modes, *insert* (to add data at the cursor) and *typeover* (to replace existing data with entered data). You cannot overwrite a NEWLINE. If the cursor in *typeover* mode encounters a NEWLINE character, the cursor mode automatically changes to *insert*, "pushing" the NEWLINE character to the right. Some keystrokes behave differently in the two modes.

When it first enters a multiple-segment field, the cursor is positioned on the first character of the first field segment, and the editing mode is set to *typeover*. The cursor movement keys are as follows:

ENTER	leaves the entire multiple-segment field, and goes to the first character of the next field.
BACKSPACE *or* LEFT ARROW	moves left one character, unless at the left edge of a field segment. From the left edge of the first segment, these either move to the first character of the preceding field, or only

beep, depending on whether INPUT WRAP is in effect. (Input wrap mode is controlled by the OPTIONS statement.) From the left edge of a lower field segment, these keys move to the rightmost intentional character of the previous field segment.

RIGHT ARROW moves right one character, unless at the rightmost intentional character in a segment. From the rightmost intentional character of the last segment, this either moves to the first character of the next field, or only beeps, depending on INPUT WRAP mode. From the rightmost intentional character of a higher segment, this moves to the first intentional character in a lower segment.

UP ARROW moves from the topmost segment to the first character of the preceding field. From a lower segment, this moves to the character in the same column of the next higher segment, jogging left, if required, to avoid editor blanks, or if it encounters a TAB.

DOWN ARROW moves from the lowest segment to the first character of the next field. From a higher segment, moves to the character in the same column in the next lower segment, jogging left if required to avoid editor blanks, or if it encounters a TAB.

TAB enters a TAB character, in insert mode, and moves the cursor to the next TAB stop. This can cause following text to jump right to align at a TAB stop. In typeover mode, this moves the cursor to the next TAB stop that falls on an intentional character, going to the next field segment if required.

The character keys enter data. Any following data shifts right, and words can move down to subsequent segments. This can result in characters being discarded from the final field segment. These keystrokes can also alter data:

CONTROL-A switches between typeover and insert mode.

CONTROL-X deletes the character under the cursor, possibly causing words to be pulled up from subsequent segments.

CONTROL-D deletes all text from the cursor to the end of the multiple-line field (not merely to the end of the current field segment).

CONTROL-N inserts a NEWLINE character, causing subsequent text to align at the first column of the next segment of the field, and possibly moving words down to subsequent segments. This can result in characters being discarded from the final segment of the field.

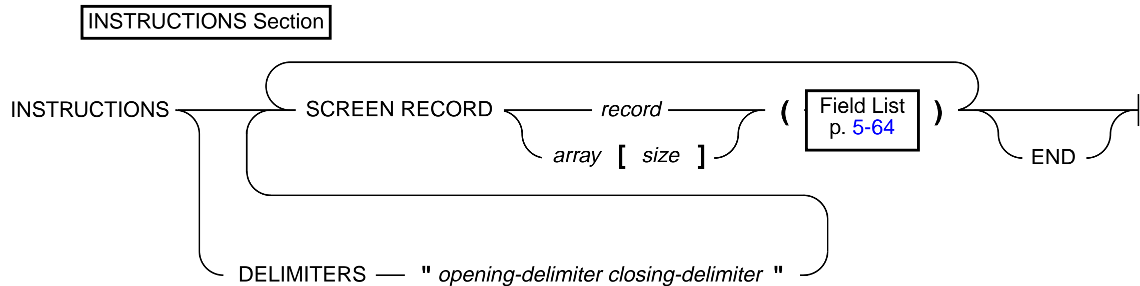
Non-WORDWRAP Displays

The appearance of a character value on the screen can vary, depending on whether or not it is displayed in a multiple-segment WORDWRAP field. For instance, if a value that was entered using WORDWRAP is displayed without this attribute, words will generally be broken, not wrapped, and TAB and NEWLINE characters will be displayed as question (?) marks. These differences do not represent any loss of data, but only a different mode of display. *(*You can view this effect, for example, if you also have INFORMIX-SQL installed on your system, and you use the **Query Language Menu** to display character data values that were entered using WORDWRAP.*)*

If a value prepared under the multiple-line editor is again edited without WORDWRAP, however, some formatting may be lost. For example, a user might type over a TAB or NEWLINE character, not realizing what it was. Similarly, a user might remove a blank from the first column of a line, and thus join a word to the last word on the previous line. These mistakes will be visible when the value is next displayed in a WORDWRAP field or in a *4GL* report that uses the WORDWRAP operator.

INSTRUCTIONS Section

The INSTRUCTIONS section is the optional final section of a form specification file. You can use this section to declare non-default *screen records* and *screen arrays.* The INSTRUCTIONS section appears after the last field description (or after the optional END keyword) of the ATTRIBUTES section.



array	is the *4GL* identifier for the screen array. (It is also the name of the screen record that comprises each line of the array.)
closing-delimiter	is the closing field delimiter.
opening-delimiter	is the opening field delimiter.
record	is the *4GL* identifier that you declare for the screen record.
size	is a literal integer (page 3-340), enclosed in square ([ ]) brackets, to specify the number of screen records in the screen array.

The END keyword is optional and provides compatibility with earlier Informix products.

Screen Records

A *screen record* is a group of fields that screen-interaction statements of the INFORMIX-4GL program can reference as a single object. By establishing a correspondence between a set of screen fields (the screen record) and a set of *4GL* variables (typically a program record), you can pass values between the program and the fields of the screen record. In many applications, it is convenient to define a screen record that corresponds to a *row* of a database table.

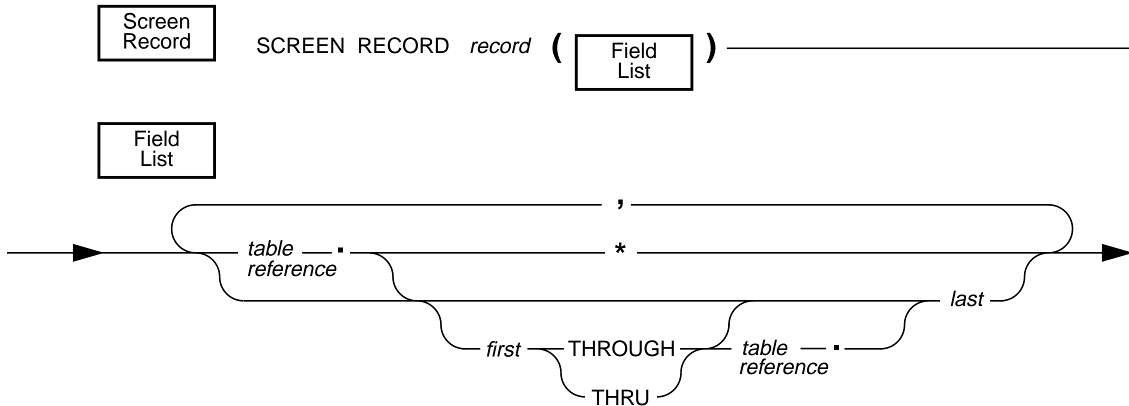
Default Screen Records

INFORMIX-4GL recognizes *default screen records* that consist of all the screen fields linked to the same database table within a given form. **FORM4GL** automatically creates a *default record* for each table that is used to reference a field in the ATTRIBUTES section. The components of the default record correspond to the set of display fields that are linked to columns in that table.

The name of the default screen record is the table name (or the alias, if you declared an alias for that table in the TABLES section). For example, all the fields linked to columns of the **customer** table constitute a default screen record whose name *is* **customer**. If a form includes one or more FORMONLY fields, those fields comprise a default screen record called **formonly**.

Non-Default Screen Records

The INSTRUCTIONS section of a form specification file can declare *non-default screen records*. You use the SCREEN RECORD keywords of the INSTRUCTIONS section to declare a *name* for the screen record, and to specify a list of *fields* that are *members* of the screen record. A record declaration has this syntax:



first is a field name that you declared in the ATTRIBUTES section.

last is a field name that you declared later than *first*.

record is the *4GL* identifier that you declare for the screen record.

table reference is a table name, alias, or synonym (or FORMONLY keyword).

The *field name* is the SQL identifier of a database column linked to the field, unless you specify FORMONLY as the table reference.

The *record name* of a non-default screen record can have up to 50 characters, and must comply with the rules for **4GL** identifiers (page 2-9).

Like the name of a screen field, the identifier of a screen record must be unique within the form, and has a scope that is restricted to when its form is open. Statements can reference *record* only when the screen form that includes it is being displayed. **FORM4GL** returns an error if *record* is the same as the name or alias of a table in the TABLE section.

The List of Member Fields

The fields within a screen record are called *members* of the record. The list of member fields must be enclosed within a pair of parenthesis (`( )`) symbols. Use comma (,) symbols to separate elements of the list of field names.

You *must* specify the *table* qualifier if the field name is not unique among the fields in the ATTRIBUTES section, or if table is a *required* alias (as described in the description of the TABLES section, page 5-18). Otherwise, *table* is optional, but including it may make the form specification file easier to read.

A screen record can include screen fields whose identifiers have different *table* specifications (including the FORMONLY keyword). You can use the notation *table.** to include default screen records in the list of fields:

```
SCREEN RECORD worlds_record
      (items.*, customer.*, state.code, FORMONLY.total)
```

Here the * symbols represents all of the fields *in the form* that the ATTRIBUTES section associated with columns in the **items** and **state** tables. These fields do *not* necessarily correspond to all of the columns in these tables, unless the form includes fields that are linked to all of the columns.

You can use the keyword THRU to specify consecutive fields, in the order of their listing in the ATTRIBUTES section from *field name 1* to *field name2*, inclusive. (The keyword THROUGH is a synonym for THRU.) For example, the following instruction creates a screen record called **address** from fields linked to some columns of the **customer** table. This record can simplify *4GL* statements to update customer address and telephone data.

```
SCREEN RECORD address
      (customer.address1 THRU customer.phone)
```

The order of fields in the portion of a screen record specified by the *table.** or THRU notation is the order of the *field names* within the ATTRIBUTES section.

Screen Arrays

A *screen array* is usually a repetitive array of fields in the screen layout, each containing identical groups of screen fields. Each "row" of a screen array is a screen record. Each "column" of a screen array consists of fields with the same field tag in the SCREEN section of the form specification file.

You must declare screen arrays in the INSTRUCTIONS section, using syntax like the syntax described for a screen record in the previous section, but with an additional parameter to specify the *number of screen records* in the array:

Screen Array

SCREEN RECORD —— *array* **[** *size* **]** —— **(** — Field List p. 5-64 — **)** ——|

array is the *4GL* identifier for the screen array. (It is also the identifier of the screen record that comprises each line of the array.)

size is a literal integer (page 3-340), enclosed in square (`[ ]`) brackets, to specify how many screen records are in the screen array.

The *size* should be the number of lines in the logical form where the set of fields that comprise each screen record is repeated within the screen array. For example, a SCREEN section might represented a screen array like this:

```
SCREEN
{
    CARRIER     FLIGHT     ARRIVES    DEPARTS
   [ f00001]  [f00002]  [ f0003]  [ f0004]
   [ f00001]  [f00002]  [ f0003]  [ f0004]
   [ f00001]  [f00002]  [ f0003]  [ f0004]
}
```

This requires a *size* of **[3]**. Except for the *size* parameter, syntax for specifying the identifier and the field names of a screen array is the same as for a simple screen record (page 5-64). Unlike *4GL* program arrays, which can have up to 3 dimensions, every 4GL screen array has exactly one dimension.

The next example declares an array of six records, each of which includes two default screen records, namely the **manufact.*** and **state.*** screen records:

```
SCREEN RECORD mant_array [6]
      (manufact.*, state.*, cust_calls.user_id,
FORMONLY.delta)
```

To illustrate the declaration of a typical screen array in more detail, consider the following fragment of a form specification file:

```
SCREEN
{
...
Item 1 [p     ][q     ][u     ][t       ]
Item 2 [p     ][q     ][u     ][t       ]
Item 3 [p     ][q     ][u     ][t       ]
Item 4 [p     ][q     ][u     ][t       ]
Item 5 [p     ][q     ][u     ][t       ]
}
TABLES orders items stock
ATTRIBUTES
...
p = stock.stock_num;
q = items.quantity;
u = stock.unit_price;
t = items.total_price;
...
INSTRUCTIONS
SCREEN RECORD sc_items[5] (stock.stock_num,
          items.quantity, stock.unit_price,
          items.total_price)
```

The **sc_items** screen array has five rows and four columns, and includes fields linked to columns from two database tables. Rows are numbered from 1 to 5. The screen record that follows the display label **Item 3** in the screen layout, for example, can be referenced as **sc_items[3]** in a *4GL* statement.

If there are no other columns of the **items** table in the form, the default screen record **items** contains two fields, corresponding to the **items.quantity** and **items.total_price** fields that are linked to columns of the **items** table.

If a screen array contains a default screen record, you can reference its fields in specific lines of the screen array (such as **items[5]** for the **q** and **t** fields in the last line), as if you had declared an array of records linked to that table.

You can reference *array-name* in the DISPLAY, DISPLAY ARRAY, INPUT, INPUT ARRAY, and SCROLL statements of INFORMIX-4GL, but only when the screen form that includes the screen array is the current form.

Screen records and screen arrays can display program records. If the fields in the screen record have the same sequence of data types as the columns in a database table, you can use the screen record to simplify *4GL* operations that pass values between program variables and rows of the database.

Field Delimiters

You can change the delimiters that **INFORMIX-4GL** uses to enclose fields when the form appears on the screen from brackets ([ ]) to any other printable character, including blank spaces. The DELIMITERS instruction tells **INFORMIX-4GL** what symbols to use as field delimiters when it displays the form on the screen. The opening and closing delimiter marks must be enclosed within quotation (") marks.

The following specifications display < and > as opening and closing delimiters of screen fields:

```
INSTRUCTIONS
    DELIMITERS "<>"
END
```

Each delimiter occupies a space, so two fields on the same line are ordinarily separated by at least two spaces. If you want only one space between consecutive screen fields, follow these two steps:

1. In the SCREEN section, substitute a vertical bar (|) for paired back-to-back ( ][ ) brackets that separate adjacent fields.
2. In the INSTRUCTIONS section, define some symbol as both the beginning and ending delimiter. For example, you could specify `"| |"` or `"/ /"` or `": :"` or `"  "` (blanks).

The following specifications substitute | for][between adjacent fields in the same line of the screen layout, and display a colon (:) as both the opening and closing delimiter:

```
SCREEN
   {
          . . .
   Full Name-[f011       |f012       ]
          . . .
   }
          . . .
INSTRUCTIONS
   DELIMITERS "::"
```

Here the fields whose tags are **f011** and **f012** will be displayed as:

```
         Full Name-:           |           :
```

If you substitute blanks for colons as DELIMITERS symbols, field boundaries are not marked (or are only marked if they have attributes that contrast with the surrounding background).

Note: **FORM4GL** *requires brackets ([ ]) in the SCREEN section of a form specification file, regardless of any DELIMITERS instruction.*

Default Attributes

Field attributes can also be specified in two special tables in the database, **syscolval** and **syscolatt**. These tables are maintained by the **upscol** utility, as described in Appendix B. **FORM4GL** searches these tables for default validation and display attribute specifications. It applies these to form fields whose names match the names of the specified database columns, or that reference these columns in the DISPLAY LIKE or VALIDATE LIKE attribute specifications.

FORM4GL adds the attributes from these tables to any attributes that are listed in the form specification file. In case of conflict, attributes from the form specification file take priority. 4GL applies the resulting set of field attributes during execution of INPUT and INPUT ARRAY statements (by using **syscolval**), and during execution of DISPLAY and DISPLAY ARRAY statements (by using **syscolatt**). The schema of each of these tables follows:

syscolval		**syscolatt**	
tabname	char(18)	**tabname**	char(18)
colname	char(18)	**colname**	char(18)
attrname	char(10)	**seqno**	serial
attrval	char(64)	**color**	smallint
		inverse	char(1)
		underline	char(1)
		blink	char(1)
		left	char(1)
		def_format	char(64)
		condition	char(64)

Here **tabname** and **colname** are the names of the table and column to which the attributes apply. Here **colname** cannot be a BYTE nor TEXT column. Valid values for the **attrname** and **attrval** columns in **syscolval** are these:

attrname	**attrval**
AUTONEXT	YES, NO (the default)
COMMENTS	as in this chapter
DEFAULT	as in this chapter
INCLUDE	as in this chapter
PICTURE	as in this chapter
SHIFT	UP, DOWN, NO (the default)
VERIFY	YES, NO (the default)

The **color** column in **syscolatt** stores an integer that describes color (for color terminals) or intensities (for monochrome terminals).

The next table shows the displays specified by each value of **color**, and the correspondence between default color names, number codes, and intensities:

Number	**Color Terminal**	**Monochrome Terminal**
0	White	Normal
1	Yellow	Bold
2	Magenta	Bold
3	Red	Bold†
4	Cyan	Dim
5	Green	Dim
6	Blue	Dim†
7	Black	Dim

The background for colors is BLACK in all cases. The † signifies that, if the keyword BOLD is specified as the attribute, the field is displayed as RED on a color screen; or, if the keyword DIM is specified as the attribute, the field is displayed as BLUE on a color screen.

The values for **inverse**, **underline**, **blink** and **left** are `Y` (yes) and `N` (no). The default for each of these columns is `N`, that is, normal display (bright characters in a dark field), no underline, steady font, and right-justified numbers. Which of these attributes can be displayed simultaneously with the color combinations or with each other is terminal-dependent.

The **def_format** column takes the same string that you would enter for the FORMAT attribute in a screen form. Do not use quotation marks.

The **condition** column takes string values that are a restricted set of the WHERE clauses of a SELECT statement, except that the WHERE keyword and the column name are omitted. INFORMIX-4GL assumes that the value in the column identified by **tabname** and **colname** is the subject of all comparisons.

Examples of valid entries for the **condition** column follow:

```
<= 100 MATCHES "[A-M]*"          BETWEEN 101 AND 1000
IN ("CA", "OR", "WA")            >= 1001 NOT LIKE "%analyst%"
```

The VALIDATE statement (page 3-278) compares the members of a program record or variable list to the validation rules in **syscolval**. The INITIALIZE statement (page 3-125) can read the default values in **syscolval** for a list of columns, and assign these values to a corresponding list of 4GL variables.

Some statements (including CONSTRUCT, DISPLAY, DISPLAY ARRAY, ERROR, INPUT, INPUT ARRAY, MESSAGE, PROMPT, OPEN WINDOW, and OPTIONS) support an ATTRIBUTE clause (page 3-290) that can specify color and intensity attributes. These attributes are also supported by the **syscolatt** table and by the COLOR keyword in the ATTRIBUTES section:

CYAN	=	DIM	GREEN	=	DIM	REVERSE	=	REVERSE
BLUE	=	DIM	YELLOW	=	BOLD	UNDERLINE	=	UNDERLINE
MAGENTA	=	BOLD	RED	=	BOLD	BLINK	=	BLINK
WHITE	=	NORMAL	BLACK	=	DIM			

Here the "=" symbol indicates how monochrome terminals interpret color keywords. On color terminals, NORMAL is displayed as WHITE; BOLD as RED; and DIM as BLUE.

You can override default attributes in **syscolatt** by assigning other attributes in the form specification file, or in the ATTRIBUTE clause of the CONSTRUCT, DISPLAY, DISPLAY ARRAY, INPUT, or INPUT ARRAY statement. If the current *4GL* statement is one of these, and includes an ATTRIBUTE clause, then the field displays only the attributes that are specified in that clause. For example, if a column is designated as RED and BLINK in **syscolatt**, or in the form specification file, and your *4GL* program executes the statement

```
DISPLAY . . . ATTRIBUTE BLUE
```

the field has only the BLUE attribute, not blinking BLUE. If an ATTRIBUTE clause is present in the currently executing statement, there is no implicit carry-over of display attributes from the compiled form (except FORMAT).

Precedence of Field Attribute Specifications

INFORMIX-4GL uses these rules of precedence (highest to lowest) to resolve any conflicts among multiply-defined display attribute specifications:

1. The ATTRIBUTE clause of the current *4GL* statement.
2. The field descriptions in the ATTRIBUTES section of the current form.
3. The default attributes specified in the **syscolatt** table of any fields linked to database columns. To modify the **syscolatt** table, use the **upscol** utility. For information on using this utility, see Appendix B.
4. The ATTRIBUTE clause of the most recent OPTIONS statement.
5. The ATTRIBUTE clause of the current form in the most recent DISPLAY FORM statement.
6. The ATTRIBUTE clause of the current 4GL window in the most recent OPEN WINDOW statement.

Default Attributes in an ANSI-Compliant Database

In a database that is not ANSI-compliant, the default screen attributes and validation criteria that you specify with the **upscol** utility are stored in two tables, **syscolval** and **syscolatt**. If these tables specify default values or attributes for a database column, those defaults are available to every user of a form that references the column.

In an ANSI-compliant database, however, the separate *owner.***syscolval** and *owner.***syscolatt** tables are created for each user of the **upscol** utility. These tables store the default specifications of that individual user. Which set of tables is used by **FORM4GL** depends on the nature of the request.

If the TABLES section specifies a table alias for *owner.table*, **FORM4GL** uses the **upscol** tables of the owner of *table*. If that user owns no **upscol** tables, no defaults are assigned to fields associated with that table alias. If the TABLES section of the form does not specify a table alias that includes the owner of a database table, the **upscol** tables owned by the user running **FORM4GL** are applied to fields associated with that database table, unless the user owns no **upscol** tables. In the ATTRIBUTES section, field descriptions of the forms

field-tag = . . . DISPLAY LIKE *table.column*
field-tag = . . . VALIDATE LIKE *table.column*

use **upscol** tables (if they exist) owned by whoever runs **FORM4GL**, unless *table* is an alias that specifies a different owner. If *table* is an alias for *owner.table*, **FORM4GL** uses the **upscol** tables of the owner specified by *table*, if they exist. If no **upscol** tables exist, then the DISPLAY LIKE and

VALIDATE LIKE attributes have no effect. If *owner* is not the correct owner, the compilation fails and an error message is issued. See also the INITIALIZE (page 3-125) and VALIDATE (page 3-278) statements.

Creating and Compiling a Form

For your *4GL* program to work with a screen form, you must create a form specification file that conforms to the syntax described earlier in this chapter, and then compile the form. You can compile the form in one of two ways: from within the Programmer's Environment or at the command line. Both methods require that the database and any tables referenced in the form already exist, and that the database engine be running and able to access the database. These methods of compiling a form are described below. Also, a section on using default forms is included.

Compiling a Form Through the Programmer's Environment

To create a screen form using the Programmer's Environment (which is described in Chapter 1), you must follow these steps:

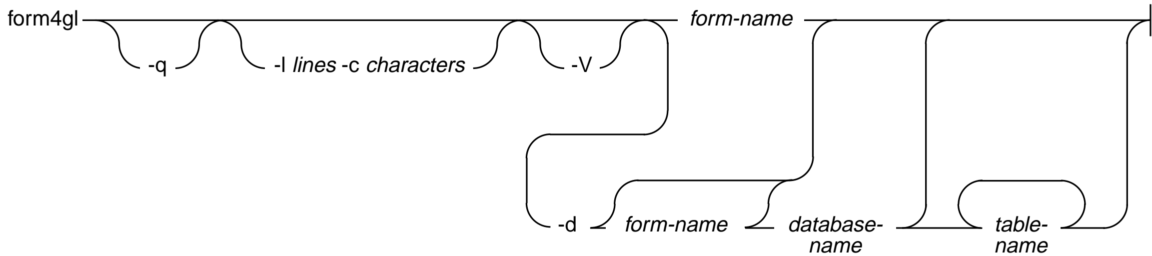
1. At the system prompt, enter `i4gl` if you have the **C Compiler Version**, or `r4gl` if you have the **Rapid Development System**.
2. Select **Form** and then **Generate** from the menu. (Alternatively, you can select the **New** option. **INFORMIX-4GL** prompts you for a form name, prompts you for an editor if you have not already selected one, and invokes that editor with an empty form specification file. Now you must enter form specifications. The **Generate** option is usually a more efficient way to create a customized form.)
3. Enter the name of the database and the name that you want to assign to the form (for example, **myform**). **INFORMIX-4GL** asks you for the names of the tables whose columns you want in your form. After you select the tables, **FORM4GL** creates a default form specification file, as well as a compiled default form, and then displays the FORM Design Menu.
4. The default form specification file formats the screen as a list of all the columns in the tables that you entered in Step 3. It does not provide any special instructions to **INFORMIX-4GL** about how to display the data. Select the **Modify** option, and **INFORMIX-4GL** presents the MODIFY FORM Screen. Select the default form specification (given as **myform** earlier), and **INFORMIX-4GL** calls a system editor to display the file. Edit the default form specification file to produce your customized screen

form and associated instructions. (You can specify an editor using the DBEDIT environment variable. This is fully explained in Appendix D.) When you save your file and quit the editor, you return to the MODIFY FORM Menu.

5. Select **Compile**. If your form specification file successfully compiles, **FORM4GL** creates a form file with the extension **.frm** (for example, **myform.frm**). Go on to Step 7. If your form specification file does not compile successfully, go on to Step 6.
6. Select the **Correct** option from the COMPILE FORM Menu. **INFORMIX-4GL** again calls your editor to display the form specification file, with the compilation errors marked. When correcting your errors, you need not delete the error messages. **INFORMIX-4GL** does that for you. Save the file and go to Step 5.
7. Save your form specification file with the **Save-and-exit** option.

Compiling a Form Through the Operating System

The **FORM4GL** command line has the following syntax:



characters	is an integer that specifies the width of the form in characters. (The default is the number of characters in the longest line of the screen layout, as specified in the SCREEN section.)
form-name	is the name of the form specification file (without the **.per** extension).
lines	is an integer that specifies the height of the form in lines of characters that the terminal can display. (The default is 24.)

Use the **-v** option to have the compiler verify that the screen fields are as wide as any corresponding character fields specified in the ATTRIBUTES section. Use the **-d** option to generate a default form specification file. When you use this option, the compiler prompts you for the names of your form file, database, and tables. For more information, see the next section, "Default Forms."

To create a customized screen form directly from the operating system, follow these steps:

1. Create a default form specification file by entering the command

   ```
   form4gl -d
   ```

   at the operating system prompt. **FORM4GL** asks for the name of your form specification file, the name of your database, and the name of a table whose columns you want in your form. It continues to ask for another table name until you enter a RETURN for the name of a table. **FORM4GL** then creates a default form specification file and appends the extension **.per** to its name. It also creates a compiled default form with the extension **.frm**.
2. Use the system editor to modify the default form specification file to meet your specifications. If, as an alternative, you create a new form specification file and skip Step 1, be sure to give the filename the extension **.per**.
3. Enter a command of the form:

   ```
   form4gl myform
   ```

   Here **myform** is the name of your form specification file (without the **.per** extension).

   If the compilation is successful, **FORM4GL** creates a compiled form file called **myform.frm** and you are finished creating your customized screen form. If not, **FORM4GL** instead creates a file named **myform.err**, and you need to go on to Step 4.
4. Review the file **myform.err** to discover the compilation errors. Make corrections in the file **myform.per**. Go to Step 3.

Default Forms

For many applications, it is convenient to create a *default form*, and then edit this to satisfy your specific application requirements. When you create a default form, you must specify its filename, a database name, and the name of at least one table whose columns are to be linked to fields in the form.

The *width* of a display field is the number of characters that can be placed between the delimiters. In a default form specification, **FORM4GL** assigns lengths to fields according to the declared data type of the column:

Data Type	Default Field Width (in characters)
BYTE	12.
CHAR	MIN (57, *n*), for *n* the length from the data-type declaration.
DATE	10.
DATETIME	From 2 to 25, as implied in data-type declaration. Each *unit of time* = 2 (except YEAR and FRACTION); every separator = 1.
DECIMAL	(2 + *m*), for *m* the precision from the data-type declaration.
FLOAT	14.
INTEGER	11.
INTERVAL	From 3 to 25 (as implied in data-type declaration, plus one).
MONEY	(3 + *m*), for *m* the precision from the data-type declaration.
SMALLINT	6.
SMALLFLOAT	14.
TEXT	12.
VARCHAR	MIN (57, *n*), for *n* the maximum length from the declaration.

SERIAL columns are linked to INTEGER fields. Field length is not directly related to the data display in BYTE and TEXT fields, both of which require the PROGRAM attribute to invoke an external program or editor. The *4GL* form can display as many TEXT characters as fit in the field, and displays the string "`<BYTE value>`" in a BYTE field. (For details of displaying blob values, see the description of the PROGRAM field attribute on page 5-50.)

If you edit a default form, make sure that the fields are wide enough to accommodate the widest value that might be entered or displayed.To prevent INFORMIX-4GL from truncating displayed data, follow these rules:

- Make character fields as wide as the corresponding database column. You can use multiple-segment fields to display long strings (page 5-26).
- Make number, DATETIME, and INTERVAL fields wide enough to accommodate the largest displayed value.

Default field tags like **f000** are assigned to the first display field, **f001** to the second, and so on, by **FORM4GL**. It assigns a field tag like **a0** to any two- or three-character field that cannot accommodate a four-character default field tag. Up to 26 single-character fields can be assigned the single-characters **a**, **b**, **c**, and so forth, as default field tags.

The default screen layout has as many lines as the number of columns in the tables. Each line of the screen layout contains a single field, beginning in the 20th character position. **FORM4GL** uses column names as default field labels, appearing to the left of each field. The next example shows a default form that is based only on the **customer** table of the **stores2** database:

```
database stores
screen size 24 by 80
{
customer_num       [f000       ]
fname              [f001           ]
lname              [f002           ]
company            [f003                ]
address1           [f004                ]
address2           [f005                ]
city               [f006           ]
state              [a0]
zipcode            [f007 ]
phone              [f008              ]
}
end
tables
customer
attributes
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state;
f007 = customer.zipcode;
f008 = customer.phone;
end
```

If the number of fields is greater than (*lines* - 4), you must edit the default file, either to increase the *lines* value after the SIZE keyword (if the screen size permits this), or else to reduce the number of lines in the screen layout.

Using PERFORM Forms in 4GL

The syntax of forms that work with the **FORM4GL** is different in several significant ways from the syntax of PERFORM, the screen form generation utility of INFORMIX-SQL. You can use PERFORM forms with 4GL, but you must first recompile them using **FORM4GL**. In addition, not all PERFORM features are operative. You must also use 4GL user-interaction statements like OPEN FORM, OPEN WINDOW, INPUT, DISPLAY FORM, CLEAR FORM, or CONSTRUCT to write a "form driver" to support data entry or data display through the 4GL form.

If you have designed forms for the PERFORM screen transaction program of INFORMIX-SQL, you need to know how those forms behave when used with 4GL. The following features differ between PERFORM and 4GL:

- Only the DELIMITERS keyword in the INSTRUCTIONS section of a PERFORM form is supported by 4GL. Other keywords in that section are ignored. To support other INSTRUCTIONS features of PERFORM requires coding in your 4GL program. (See the BEFORE and AFTER clauses of the INPUT statement.)
- 4GL does not support multiple-page forms (those with more than one screen layout); these produce undesirable overlays. (Use multiple *4GL* forms to produce the effects of forms that have several pages.)
- There is no concept of "current table" in 4GL. A single INPUT or INPUT ARRAY statement allows you to enter data into fields that correspond to columns in different tables.
- Joins defined in the PERFORM form are ignored in 4GL. You can associate two field names with the same field tag, using the same notation as in a PERFORM join, but no join is effected. On the other hand, you can create more complex joins and look-ups in 4GL using the full power of SQL.
- The PERFORM attributes LOOKUP, NOUPDATE, QUERYCLEAR, RIGHT, and ZEROFILL are inoperative in 4GL. The DISPLAY, DISPLAY ARRAY, DISPLAY FORM, MESSAGE, and OPEN WINDOW statements of 4GL all ignore the INVISIBLE attribute (page 5-46).
- The *conditions* of a COLOR attribute cannot reference other field tags nor aggregate functions.
- The default attributes listed in **syscolval** and **syscolatt** do not apply to your PERFORM forms, unless you recompile the forms with **FORM4GL**.

Chapter

6

INFORMIX-4GL Reports

Output from 4GL Programs 3

Features of 4GL Reports 3

Producing 4GL Reports 4
- The Report Driver 5
- The REPORT Definition 5
  - The Report Prototype 6
  - Components of the Report Definition 7

DEFINE Section 8

OUTPUT Section 9
  - The BOTTOM MARGIN Clause 11
  - The LEFT MARGIN Clause 12
  - The PAGE LENGTH Clause 12
  - The REPORT TO Clause 13
  - The RIGHT MARGIN Clause 14
  - The TOP MARGIN Clause 16
  - The TOP OF PAGE Clause 17

ORDER BY Section 18
  - The Sort List 19
  - The Sequence of Execution of GROUP OF Control Blocks 20
  - The EXTERNAL Keyword 22

FORMAT Section 23
  - EVERY ROW 24

FORMAT Section Control Blocks 27
AFTER GROUP OF 29
The Order of Processing AFTER GROUP OF Control Blocks 29
The GROUP Keyword in Aggregate Functions 30
BEFORE GROUP OF 31
The Order of Processing BEFORE GROUP OF Control Blocks 31
FIRST PAGE HEADER 33
Displaying Titles and Headings 33
Restrictions on the List of Statements 34
ON EVERY ROW 34
Group Control Blocks 35
ON LAST ROW 36
PAGE HEADER 37
PAGE TRAILER 38
Restrictions on the List of Statements 38

Statements in REPORT Control Blocks 39
NEED 40
PAUSE 41
PRINT 42
The FILE Option 44
The Character Position 44
The Expression List 45
Aggregate Report Functions 46
The ASCII Operator 47
The COLUMN Operator 48
The LINENO Operator 48
The PAGENO Operator 48
The SPACE or SPACES Operator 49
The WORDWRAP Operator 50
SKIP 52
Restrictions on SKIP Statements 52

Output from 4GL Programs

INFORMIX-4GL offers several features to output values from an SQL database or from 4GL program variables, including the following:

- Output of unformatted database rows to an ASCII file by using the UNLOAD statement (page 3-274).
- Direct screen output by using DISPLAY statements to display values that the SELECT statement has retrieved from the database and stored in 4GL program variables. (The SELECT statement is described in the *Informix Guide to SQL: Reference*.)
- Output to a 4GL form (as described in Chapter 5) through the DISPLAY or DISPLAY ARRAY statements.
- Output to a reserved line (page 3-93) of 4GL through the ERROR, PROMPT, MENU, or MESSAGE statement, or the COMMENTS attribute (page 5-36).
- Output of TEXT or BYTE values to an external editor that you specify through the PROGRAM field attribute of a 4GL form (page 5-50).
- Output to the screen (page 6-14) or to a file from a 4GL report.

This chapter describes 4GL reports, the method of producing output that offers the greatest formatting flexibility.

Features of 4GL Reports

For relational database management applications, INFORMIX-4GL includes a general-purpose *report writer* that supports the following features:

- Full control over page layout for your 4GL report. This includes first-page headers that differ from headers on subsequent pages, page trailers, columnar data presentation, special formatting before and after groups of sorted data, and data-dependent conditional formatting.
- Facilities for creating the report either from the rows returned by a cursor or from input records assembled from any other source, such as output from several different SELECT statements.

- Control blocks for manipulating data returned by the cursor on a row-by-row basis, either before or after the row is formatted by the report.
- Aggregate functions that enable you to calculate and display frequencies, percentages, sums, averages, maxima, and minima.
- The USING operator and other built-in 4GL functions for formatting and displaying information in the report.
- The WORDWRAP operator to format long character strings that occupy multiple lines of output from the report.
- The option to update the database or execute any sequence of SQL and other 4GL statements while writing a report, if the intermediate values calculated by the report meet specified criteria. For example, you could even write an alert message containing a second report.

This chapter describes how to write REPORT definitions, and how to use them to format data sets.

Producing 4GL Reports

Many relational database management applications are designed to produce a *report* that contains information from the database. A 4GL report can arrange and format the data according to your instructions, and display the output on the screen, or send it to a printer, or store it as a file for future use. To write a report, a 4GL program must include two distinct components:

- The *report driver* specifies what data the report includes.
- The REPORT routine (also called the *report definition*) formats the data.

The *report driver* (also called the *calling routine*) retrieves the specified rows from a database, stores their values in program variables, and sends these, one input record at a time, to the REPORT definition. After the last input record has been received and formatted, 4GL calculates any aggregate values (such as frequency counts, sums, or averages) that are based on all the data, and then sends the entire report to some output device.

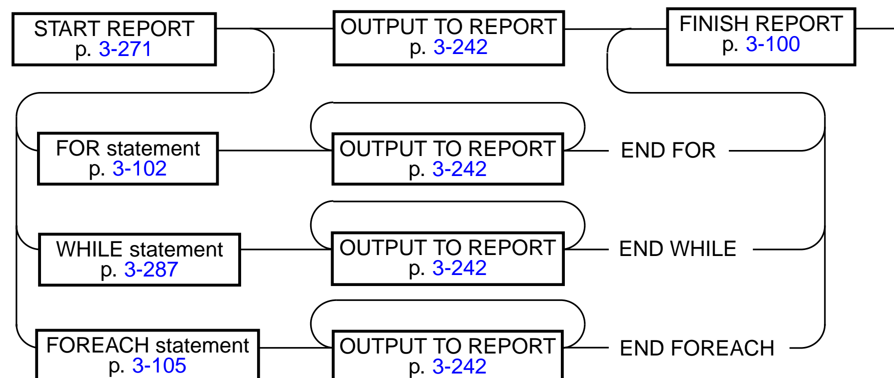
By separating the two tasks of *data retrieval* and *data formatting* in this way, 4GL simplifies the production of recurrent reports, as well as the application of the same report format to different data sets.

The Report Driver

The report driver invokes the report, retrieves data, and sends the data (as *input records*) to be formatted by the REPORT program block. A report driver can be part of the MAIN program block, or in one or more 4GL functions. It requires special-purpose statements to interface with the REPORT definition:

- START REPORT
- OUTPUT TO REPORT
- FINISH REPORT

These elements of the report driver can appear in different program blocks, but they are typically embedded within a FOR, FOREACH, or WHILE loop:



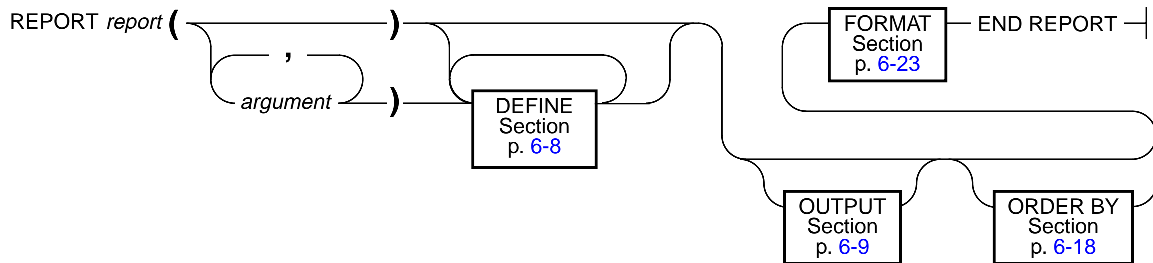
- Uses START REPORT (page 3-271) to initialize the report.
- Begins a FOR, FOREACH, or WHILE loop to control the repeated fetching of rows, and the creation of input records for storing the retrieved data.
- Uses OUTPUT TO REPORT (page 3-242) to pass input records to the report.
- Terminates the loop (with the END FOR, END FOREACH, or END WHILE keywords) after all the desired values have been passed to the report.
- Uses FINISH REPORT (page 3-100) to execute any ON LAST ROW control block and to activate any two-pass report processing (page 6-22).

The REPORT Definition

The REPORT definition formats input records. Like the FUNCTION or MAIN statement, it is a program block. It can contain control blocks with statements for producing headers, footers, and calculating aggregate values.

From the report driver, the REPORT definition receives data in sets called *input records*. These can include program records, but other data types (including built-in and programmer-defined *classes*) are also supported. Each input record is formatted and printed, as specified by control blocks and statements within the REPORT definition. Most 4GL statements and functions can be included in a REPORT definition, and certain specialized statements and operators for formatting output can appear *only* in a REPORT definition.

The REPORT program block has the following syntax:



argument is the name of a formal argument that the driver passes to the report. The list of identifiers is called the *argument list*. (You can include arguments of the RECORD data type in this list, but you cannot append the .* symbols here to the name of the record.)

report is the 4GL identifier that you declare here for the report.

To format input records, a typical REPORT definition performs these actions:

- Specifies a REPORT *prototype* to declare the report name and the names of the formal arguments of the input records that the report will format.
- Uses a DEFINE section to declare formal arguments and local variables.
- Uses control blocks within the FORMAT section to produce headers, footers, and formatted output of the data in the input records.
- Terminates processing the data with the END REPORT keywords.

In a typical RDBMS application, the input records that the report formats contain values that SQL statements retrieved from the database, but a 4GL report can also process input records that were not derived from the database.

The Report Prototype

The *report* name and the *argument list* (enclosed in parentheses) are called the *report prototype*. In its syntax, it resembles a function prototype (page 3-112).

You must declare the *name* of the report and (between parentheses) the names of all the arguments that contain the data that the driver passes to the report:

```
REPORT mcbeth_report (sound,fury)
```

A formal argument cannot be an ARRAY variable, nor a RECORD variable that contains an ARRAY member. Unless the argument list is empty, its arguments must be declared in the DEFINE section (page 6-8) as local variables. You must specify a *argument list* whenever any of the conditions are true that are listed on the next page for declaring report arguments:

If you do not specify any variables in the argument list, you can print text from the control blocks, but the only data that the report can include must be contained in module variables or global variables.

Components of the Report Definition

The REPORT definition is composed of up to four sections. If any of the first three are included, they must appear in the following order:

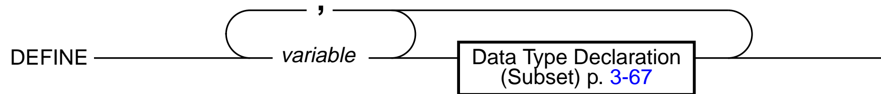
- **DEFINE Section:** This section declares the data types of local variables used within the report, and of any variables (the input records) that are passed as arguments to the report by the calling statement. Reports without arguments or local variables do *not* require a DEFINE section.
- **OUTPUT Section:** Output from the report consists of successive *pages*, each containing a fixed number of *lines* whose margins and maximum number of characters is fixed. This section can set margin and page size values, and can also specify where to send the formatted output.
- **ORDER BY Section:** This section specifies the variables on records are to be sorted. It is required if the report driver does not send sorted data to the report. The specified sort order determines the order in which 4GL processes any GROUP OF control blocks in the FORMAT section.
- **FORMAT Section:** This section is required. It specifies the appearance of the report, including page headers, page trailers, and aggregate functions of the data. It can also contain control blocks that specify actions to take before or after specific groups of rows are processed. (Alternatively, it can produce a *default report* by only specifying FORMAT EVERY ROW.)

Each of these four sections begins with the keyword for which it is named. These elements of a REPORT definition are described in sections that follow.

Line MAIN or FUNCTION, the REPORT definition must appear outside any other program block. It must begin with the REPORT statement, and must end with the END REPORT keywords. The FORMAT section is always required. You can include other sections as needed.

DEFINE Section

This section declares a data type for each formal argument in the REPORT prototype, and for any additional *local variables* that can be referenced only within the REPORT program block. The DEFINE section is required if you pass arguments to the report, or if you reference local variables in the report.



variable is the name of a local variable or formal argument of the report.

Usage

For declaring local variables, the same rules apply to the DEFINE section as to the DEFINE statement (page 3-65) in MAIN and FUNCTION program blocks. Two exceptions, however, restrict the data types of formal arguments:

- Report argument cannot be of type ARRAY.
- Report argument cannot be records that include ARRAY members.

You must include arguments in the report prototype (page 6-6), and declare them in the DEFINE section, if any of the following conditions are true:

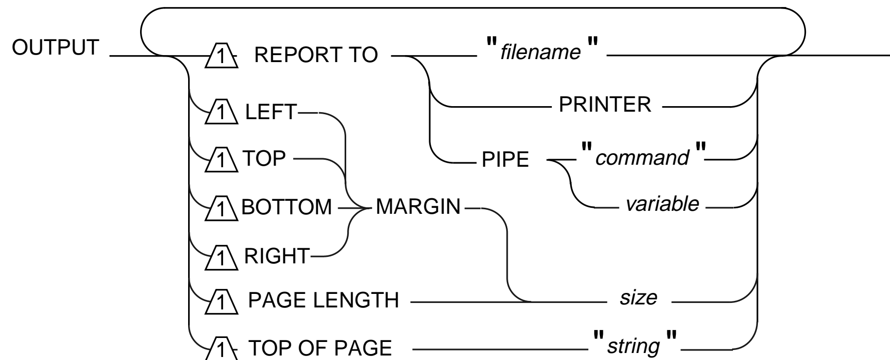
- If you specify FORMAT EVERY ROW to create a default report (page 6-24). In this case, you must pass all the values for each record of the report.
- If an ORDER BY section (page 6-18) is included. In this case, you must pass all the values that ORDER BY references, for each record of the report.
- If an aggregate that depends on all records of the report appears anywhere except in the ON LAST ROW control block (page 6-36), you must pass each of the records of the report through the argument list. Examples of aggregates dependent on all records include using GROUP PERCENT(*) (page 6-47) anywhere in a report, or using any aggregate without the GROUP keyword anywhere outside the ON LAST ROW control block.
- If you use the AFTER GROUP OF control block (page 6-29). In this case, you must pass at least the arguments that are named in that control block.

If you use the LIKE keyword (page 3-69) to specify data types indirectly, the DATABASE statement must appear before the first program block of the same module that includes the REPORT definition, as described on page 3-59.

If the DEFINE section declares a variable or argument with the same identifier as a global or module variable, that global or module variable is not visible in the report. See also the DEFINE statement (page 3-65).

OUTPUT Section

The OUTPUT section is an optional section that can specify the number of lines on each page of report output and the sizes of the margins. Without the OUTPUT section, the report uses default values to format each page.



command is a quoted string, specifying a program, shell script, or command line to receive the output from the report.

filename is a quoted string, specifying the name of a file to receive the report output. The *filename* can also include a pathname.

size is a literal integer (page 3-340) that specifies the vertical height (in lines) of the page or of the top or bottom margin, or the horizontal width (in characters) of the left or right margin.

string is a quoted string, specifying the page-eject character sequence.

variable is the name of a CHAR or VARCHAR variable that contains a shell script or command line to receive the output from the report.

Usage

This section can direct the output from the report to a *file* or to a *printer* as the default output destination. The report driver can override this default by specifying another destination in the TO clause of the START REPORT statement (page 3-271). If output goes to the printer, the TOP OF PAGE clause can specify a page-eject sequence (page 6-17) to begin each new page of report output, rather than padding each page with blank lines.

The specification after the PIPE keyword (page 6-13) identifies a program, shell script, or command line to receive the output from the report.

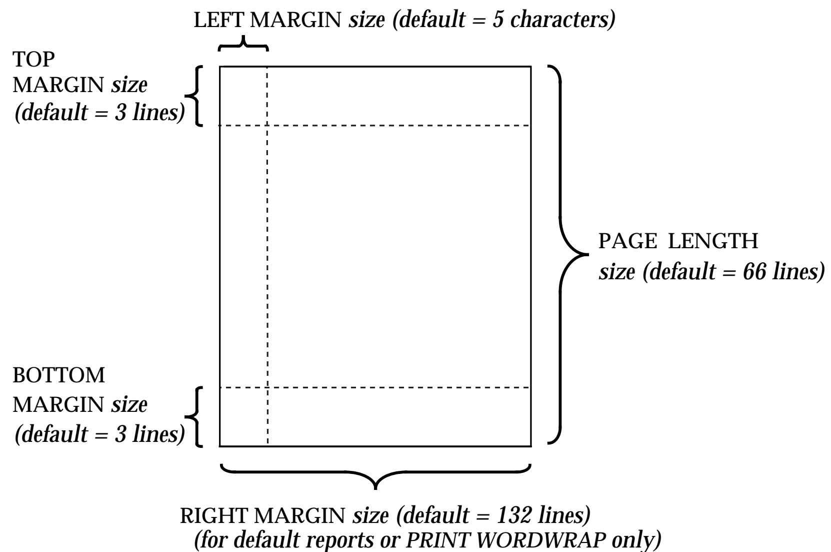
The LENGTH and MARGIN clauses specify the *height* of each page of output (see page 6-12) or the top margin (page 6-16) or bottom margin (page 6-11), or the *width* of the left margin (page 6-12) or right margin (page 6-14).

The TOP OF PAGE clause specifies a 1- or 2-character *page-eject* sequence for the default printer (page 6-17).

The OUTPUT section consists of the OUTPUT keyword, followed by one or more specifications. The OUTPUT section has the following structure

- The REPORT TO clause specifies a default destination for output. If you omit this clause, the default is to the screen (page 6-13).
- The TOP OF PAGE clause specifies the character string that causes the printer that produces a physical copy of the report to eject a page.

The next five clauses specify the physical dimensions of a 4GL report page:



- The LEFT MARGIN clause specifies how many blank spaces to include at the beginning of each new line of output. The default is 5 blank spaces.
- The RIGHT MARGIN clause specifies the maximum number of characters in each line of output, including the left margin. If you omit this but specify FORMAT EVERY ROW, the default is 132 character positions wide.

- The TOP MARGIN clause specifies how many blank lines appear above the first line of text on each page of output. The default is 3 blank lines.
- The BOTTOM MARGIN clause specifies how many blank lines follow the last line of output on each page. The default is 3 blank lines.
- The PAGE LENGTH clause specifies the total number of lines on each page, including data, the margins, and any page headers or page trailers from the FORMAT section. The default page length is 66 lines.

Sections that follow describe these OUTPUT statements in alphabetical order.

The BOTTOM MARGIN Clause

This clause sets a bottom margin for each page of the report.

```
──► BOTTOM MARGIN ──── size ──────────────────►
```

size is a literal integer (page 3-340) that specifies the non-negative vertical *height* (in lines) of the bottom margin of each page.

The bottom margin appears as *size* blank lines below any output specified by the PAGE TRAILER control block of the FORMAT section. If you do not include a BOTTOM MARGIN specification, the default bottom margin is three lines, meaning that at least three lines are left blank at the end of each page.

The following BOTTOM MARGIN specification instructs INFORMIX-4GL to continue printing to the bottom of each page.

```
OUTPUT
   REPORT TO "sendthis.out"
   TOP MARGIN 0
   BOTTOM MARGIN 0
   PAGE LENGTH 6
```

The LEFT MARGIN Clause

This clause sets the width of a left margin for each line of the report.

```
──► LEFT MARGIN ─── size ──────────────►
```

size is a literal integer (page 3-340) that specifies the non-negative *width* (in character positions) of the left margin of each page.

Output begins in the (*size* + 1) character position. Measurements indicated by arguments to the COLUMN function are always relative to the margin set by LEFT MARGIN. If you do not include a LEFT MARGIN clause, the default value for the left margin is five character positions; any output of data begins in the 6th character position.

The following LEFT MARGIN specification instructs INFORMIX-4GL to begin printing each line of the report as far to the left as possible.

```
OUTPUT
   REPORT TO "about.out"
   LEFT MARGIN 0
   PAGE LENGTH 6
```

The PAGE LENGTH Clause

This clause specifies the number of lines on each page of the report.

```
──► PAGE LENGTH ─── size ──────────────►
```

size is a literal integer (page 3-340) that specifies the non-negative *height* (in lines) of each page, including top and bottom margins.

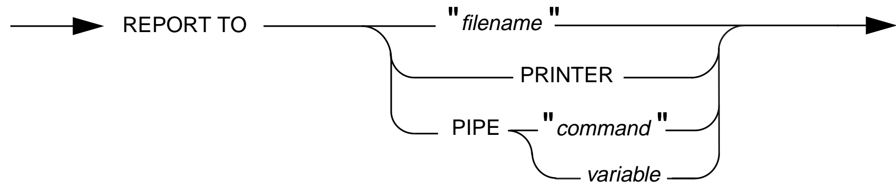
If you omit the PAGE LENGTH specification, the default page length is 66 lines. The next example specifies a PAGE LENGTH value of twenty-two lines:

```
OUTPUT
   PAGE LENGTH 22
   BOTTOM MARGIN 0
```

Depending on your font size, 22 lines is the maximum length that you can use on some systems with the PAUSE statement without causing undesirable scrolling.

The REPORT TO Clause

This clause specifies a default destination for output from the report. This destination can be a *file*, an operating system *pipe*, or the system *printer*.



command is a quoted string, specifying a program, shell script, or command line to receive the output from the report.

filename is a quoted string, containing the name of a file to receive the report output. This *filename* can also include a pathname.

variable is the name of a CHAR or VARCHAR variable that contains a shell script or command line to receive the output from the report.

If a valid START REPORT statement includes a TO clause directing output of the report to some destination, that destination takes precedence, and any REPORT TO clause in the OUTPUT section has no effect.

Sending Report Output to a Pipe

The specification after the PIPE keyword must return the name of a program, shell script, or command line that is to receive the report output. This can also include command-line arguments.

The REPORT TO PIPE option can direct the output to a program that sends the output to the correct printer, or to some other process.

The following OUTPUT section directs the report output to the **more** utility:

```
OUTPUT
   REPORT TO PIPE "more"
```

Sending Report Output to a Printer

If you specify REPORT TO PRINTER, INFORMIX-4GL sends the report to the program named in the DBPRINT environment variable. If you do not set the DBPRINT environment variable, the report is sent to your default printer. For example, the following code segment sends report output to the printer:

```
OUTPUT
   REPORT TO PRINTER
```

Sending Report Output to a File

To send the report to a printer other than the system printer, use the REPORT TO *"filename"* option to send output to a file, and then send the file to a printer of your choice. The next example of an OUTPUT section directs the report output to the **label.out** file.

```
OUTPUT
   REPORT TO "label.out"
   LEFT MARGIN 0
   TOP MARGIN 0
   BOTTOM MARGIN 0
   PAGE LENGTH 6
```

Sending Report Output to the Screen

Output is directed to the screen if both the REPORT TO clause and the TO clause of the START REPORT statement are omitted. To pause the display of the report after each screenful of output, include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. The PAUSE statements waits for the user to press the RETURN key before displaying more output. For more information on the PAUSE statement, see page 6-41.

The RIGHT MARGIN Clause

This sets the right margin for each line of a *default report* (one that specifies EVERY ROW in the FORMAT section) or of a PRINT WORDWRAP statement.

This is the syntax of the RIGHT MARGIN clause:

```
RIGHT MARGIN ———— size ————————————————|
```

size is a literal integer (page 3-340) that specifies the maximum number of characters on each line, including the left margin.

This clause sets the *right margin* by specifying a line width, in characters. The *size* is not dependent on the LEFT MARGIN, but starts its count from the left edge of the page, so that the width of the LEFT MARGIN is included in the *size* of RIGHT MARGIN. The 132-character default *size* is effective only when:

- The RIGHT MARGIN clause is omitted from the OUTPUT section, and
- The FORMAT section contains the EVERY ROW specification for a *default report* format, or else a PRINT statement with WORDWRAP is executing.

A default EVERY ROW report lists the variable names across the top of the page, and presents the data in columns beneath these headings. If there is not sufficient room between left- and right-margins to do this, INFORMIX-4GL produces a two-column output format that lists the *variable name* and the *data value* of each output record on each line of output.

The following example illustrates a RIGHT MARGIN clause. After processing the OUTPUT section, INFORMIX-4GL sets a maximum line width of 70, and does not allow text to be printed to the right of the 70th character position:

```
REPORT simple(customer)
DEFINE customer LIKE customer.*
OUTPUT
   RIGHT MARGIN 70
FORMAT
   EVERY ROW
END REPORT
```

Setting a Temporary Line Width with WORDWRAP

The PRINT statement in the FORMAT section can also include a WORDWRAP RIGHT MARGIN clause. This sets a *temporary* right margin that cannot be larger than the explicit or default right margin of the OUTPUT section. While its PRINT statement is executing, this *temporary* line width overrides the explicit or default right margin from the OUTPUT section. After the PRINT statement completes execution, the explicit or default RIGHT MARGIN *size* from the OUTPUT section is restored as the maximum line width.

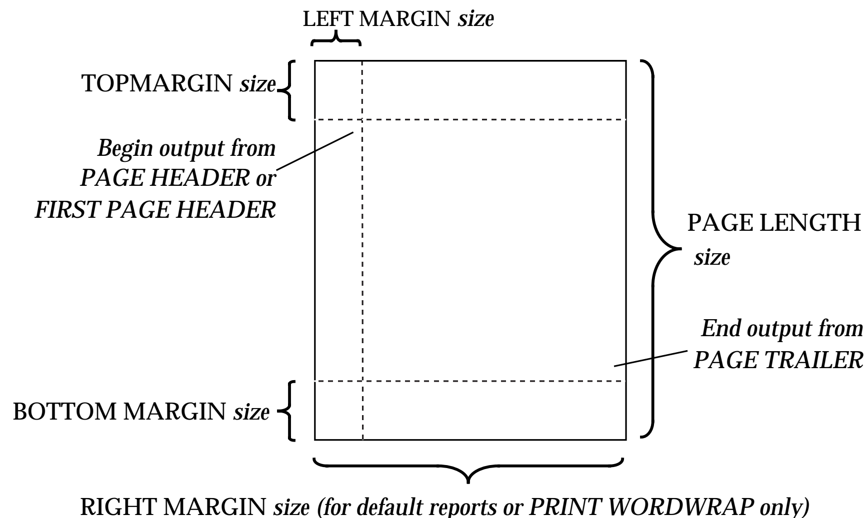
The TOP MARGIN Clause

This clause sets a top margin for each page of the report.

TOP MARGIN —— *size* ——

size is a literal integer (page 3-340) that specifies the vertical *height* (in lines) of the top margin of each page.

The number of blank lines specified as the top margin *size* appears in report output above any page header that you specify in a PAGE HEADER or FIRST PAGE HEADER control block of the FORMAT section.



If you omit the TOP MARGIN specification, then the default top margin is three lines, and any page header begins in the fourth line.

The sum of the *size* values that you specify as your top and bottom margins, plus the number of lines (if any) for the page header and trailer, represents the portion of each page that is *not* available for displaying data. Unless the page length is greater than this total, your report cannot display any records.

The following TOP MARGIN clause begin printing at the top of each page:

```
OUTPUT
   TOP MARGIN 0
   PAGE LENGTH 65
```

The TOP OF PAGE Clause

This optional clause identifies the *page-eject* character sequence for a printer.

```
──► TOP OF PAGE ── "string" ──────────►
```

string is a quoted string that begins with the page-eject character.

If you include the TOP OF PAGE clause, 4GL uses the specified page-eject character to set up new pages. For example, the TOP OF PAGE clause in the following example specifies CONTROL-L as the *page-eject* character:

```
REPORT labels_report (rl)
    DEFINE rl RECORD LIKE customer.*
OUTPUT
    TOP OF PAGE "^L"
    REPORT TO "r_out"
```

On many printers, this string is “^L”, the ASCII form-feed character. 4GL uses the first character of the string as the TOP OF PAGE character, unless it is the caret (^) character. In this case, 4GL interprets the second character as a control character. (If you are not sure of what character string to specify for a given printer, refer to the documentation of that printer.)

If the report definition includes the TOP OF PAGE clause, all page breaks in the output are initiated by using the specified page-eject character, rather than by padding with blank lines. If no TOP OF PAGE clause is included, then LINEFEED characters are used (before the page trailer) to pad each page to the proper length before each page break.

New Pages of Report Output

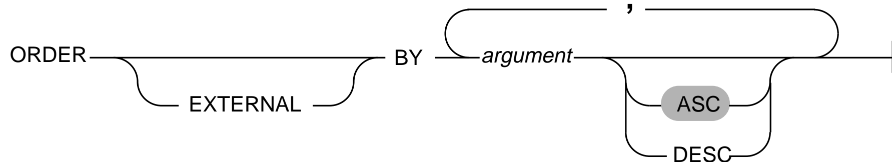
In the output from the report, 4GL includes blank line padding (or else the page-eject character, if you specify this in the *string*) to advance to the next page whenever the program causes a new page of output to be set up. New pages can be initiated by any of the following conditions:

- PRINT attempts to print on a page that is already full.
- SKIP TO TOP OF PAGE is executed.
- SKIP *n* LINES specifies more lines than are available on the current page.
- NEED specifies more lines than are available on the current page.

If you omit the TOP OF PAGE clause, 4GL fills the remaining lines of the current page with LINEFEED characters when a new page is set up.

ORDER BY Section

The ORDER BY section specifies how to sort input records, and determines the sequence of execution of GROUP OF control blocks in the FORMAT section.



argument is the name of an argument from the report prototype (page 6-6). The list of variables that you specify here is called the *sort list*.

Usage

The ORDER BY section specifies a sort list for the input records. Include this section if values that the REPORT definition receives from the report driver are significant in determining how BEFORE GROUP OF or AFTER GROUP OF control blocks will process the data in the formatted report output.

If you omit the ORDER BY section, 4GL processes input records in the order that they are received from the report driver, and processes any GROUP OF control blocks in their order of appearance in the FORMAT section. If records are not sorted in the report driver, the GROUP OF control blocks (pages 6-29 and 6-31) may be executed at random intervals (that is, after any input record), because unsorted values tend to change from record to record.

If you specify only one variable in the GROUP OF control blocks, and the input records are already sorted in sequence on that variable by the SELECT statement, then you do not need to include an ORDER BY section in the report.

The Sort List

The list of variables in the ORDER BY section specifies the order in which 4GL sorts the input records. You can only sort on the variables that appear in the argument list of the REPORT statement. The following program fragment, for example, sorts output in ascending order of **stock_tot** values:

```
REPORT r_invoice (c, stock_tot)
    DEFINE  c RECORD LIKE customer.*,
            stock_tot SMALLINT

    ORDER BY stock_tot
```

If you include more than one variable in the sort list, 4GL uses the left-to-right sequence of variables as the order of decreasing precedence. Unless the DESC keyword is specified, records are sorted in *ascending* (lowest-to-highest) order by values of the first (highest priority) variable. Records having the same value for the first variable are ordered by values of the second variable, and so on. Records with the same values on all but the last (lowest priority) variable in the sort list are ordered by that variable.

If you specify the DESC keyword, the report sorts records in *descending* order (highest-to-lowest) of values for the specified variable(s).

The next program fragment sorts records first by **zipcode**, and then within the same **zipcode** by **comp_name**, and within **comp_name** by **address1**:

```
REPORT labels_rpt(c)
    DEFINE c    RECORD LIKE custome.*
    ORDER BY c.zipcode, c.comp_name, c.address1
```

You can also sort the records by specifying a sort list in the ORDER BY clause of the SELECT statement (in the report driver). If you specify sort lists in both the report driver and the REPORT definition, the sort list in the ORDER BY section of the REPORT takes precedence.

You should use the ORDER BY clause of the SELECT statement instead of the ORDER BY section if the input records come from database rows returned by only one cursor.

Even if the REPORT definition receives records sorted by the report driver, you may wish to specify ORDER EXTERNAL BY to specify the exact order in which GROUP OF control blocks are processed. The EXTERNAL keyword (page 6-22) can prevent the input records from being sorted again.

The Sequence of Execution of GROUP OF Control Blocks

The ORDER BY section determines the order in which 4GL processes BEFORE GROUP OF and AFTER GROUP OF control blocks (pages 6-29 and 6-31). If you omit the ORDER BY section, 4GL processes any GROUP OF control blocks in the lexical order of their appearance within the FORMAT section.

If you include an ORDER BY section, and the FORMAT section contains more than one BEFORE GROUP OF or AFTER GROUP OF control block, then the order in which these control blocks are executed is determined by the sort list in the ORDER BY section. In this case, their order within the FORMAT section is not significant, because the sort list overrides their lexical order.

4GL processes all the statements in a BEFORE GROUP OF or AFTER GROUP OF control block on these occasions:

- Each time that the value of the current group variable changes.
- Each time that the value of a higher-priority variable changes.

How often the value of the group variable changes depends in part on whether the input records have been sorted:

- If the records are sorted, AFTER GROUP OF executes after 4GL processes the last record of the group of records; BEFORE GROUP OF executes before 4GL processes the first records with the same value for the group variable.
- If the records are not sorted, the BEFORE GROUP OF and AFTER GROUP OF control blocks may be executed before and after each record, because the value of the group variable may change with each record. All the AFTER GROUP OF and BEFORE GROUP OF control blocks are executed in the same lexical order in which they appear in the FORMAT section.

The following program illustrates how the ORDER BY section and the GROUP OF control blocks interact:

```
MAIN
    START REPORT sample_rpt TO "sample.out"
    OUTPUT TO REPORT sample_rpt (1,1,1)
    OUTPUT TO REPORT sample_rpt (2,2,2)
    FINISH REPORT sample_rtp
END MAIN
```

```
REPORT sample_rpt (a,b,c)
    DEFINE a,b,c, col INTEGER
    ORDER EXTERNAL BY a,b,c
    FORMAT
        FIRST PAGE HEADER
            LET col = 0
    ON EVERY ROW
            PRINT COLUMN col, "**rec**", a,b,c
    AFTER GROUP OF c
        PRINT COLUMN col, "after c"
        LET col = col - 4
    AFTER GROUP OF a
        PRINT COLUMN col, "after a"
        LET col = col - 4
    AFTER GROUP OF b
        PRINT COLUMN col, "after b"
        LET col = col -4
    BEFORE GROUP OF b
        LET col = col + 4
        PRINT COLUMN col, "before b"
    BEFORE GROUP OF a
        LET col = col + 4
        PRINT COLUMN col, "before a"
    BEFORE GROUP OF c
        LET col = col + 4
        PRINT COLUMN col, "before c"
END REPORT
```

The **sample_rpt** report in the previous example produces output in **a**, **b**, **c** order (for the BEFORE GROUP OF control blocks) and **c**, **b**, **a** order (for the AFTER GROUP OF control blocks), based on the **a**, **b**, **c** order that the ORDER BY section specifies:

```
before a
    before b
        before c
        **rec**         1          1          1
        after c
        after b
after a
before a
    before b
        before c
        **rec**         2          2          2
        after c
    after b
after a
```

If you delete or comment out the ORDER BY section, however, the resulting code would produce the following output, based on the physical sequence of variables in GROUP OF control blocks (here **c**, **a**, **b**) in the FORMAT section:

```
before c
    before a
        before b
        **rec**          1          1          1
        after b
    after a
after c
before c
    before a
        before b
        **rec**          2          2          2
        after b
    after a
after c
```

The EXTERNAL Keyword

Specify ORDER EXTERNAL BY if the input records have already been sorted by the SELECT statement. The list of variables after the keywords ORDER EXTERNAL BY control the execution order of GROUP BY control blocks.

Without the EXTERNAL keyword, the report is a *two-pass* report, meaning that 4GL processes the set of input records twice. During the first pass, it sorts the data and stores the sorted values in a temporary file in the database. During the second pass, 4GL calculates any aggregate values, and produces output from data in the temporary files.

With the EXTERNAL keyword, 4GL only needs to make a single pass through the data: it does not need to build the temporary table in the database for sorting the data. Specifying EXTERNAL to instruct 4GL not to sort the records again may result in an improvement in performance.

In the previous code example, the EXTERNAL keyword in the ORDER BY section instructs the report to accept the input records without sorting them. Without this keyword, the report needs access to a database to create its temporary table for sorting. (If no database is open and you run a two-pass report, a run-time error occurs when 4GL cannot create the temporary table.)

If the input records for your report come sequenced in the desired order (for example, from the rows returned by only one cursor), or if you want values in *descending* order, you should use the ORDER BY clause in the SELECT statement that is associated with the cursor. Then use the EXTERNAL keyword in the ORDER BY section of your report.

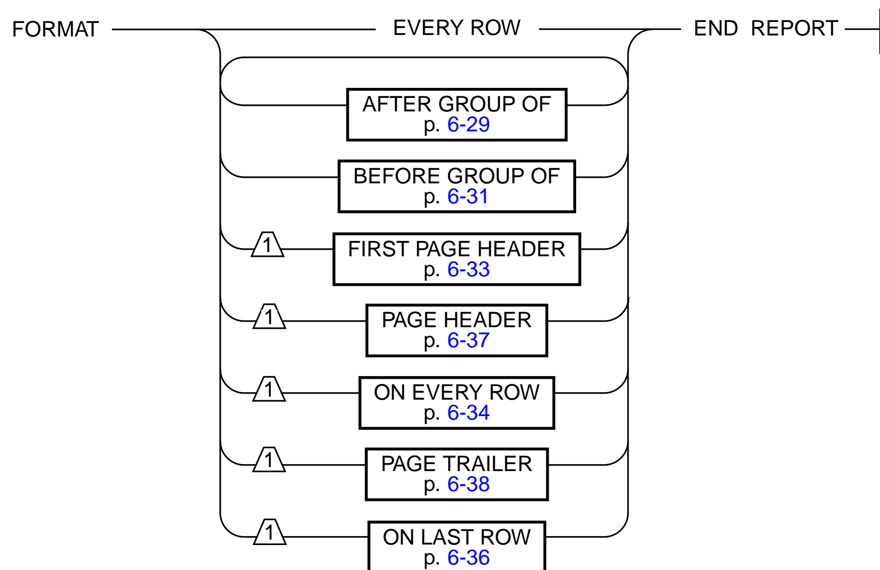
FORMAT Section

A REPORT definition must contain a FORMAT section. The FORMAT section determines what the output from the report will look like. It works with the values that are passed to the REPORT program block through the argument list, or with global or module variables in each record of the report. In a source file, the FORMAT section begins with the FORMAT keyword, and ends with the END REPORT keywords.

4GL supports two types of FORMAT sections. The simplest (a *default report*) contains only the EVERY ROW keywords (page 6-24) between the FORMAT and END REPORT keywords.

More complex FORMAT sections can contain *control blocks*, like ON EVERY ROW or BEFORE GROUP OF, containing statements to execute while the report is being processed. Control blocks can contain report execution statements (page 6-39), and other executable 4GL statements that are not SQL statements.

Sections that follow describe the syntax of default FORMAT sections, of the seven types of FORMAT section control blocks, and of the report execution statements that can appear only within a control block. The FORMAT section has the following structure:



If you do not use the EVERY ROW keywords to specify a default report, you can combine one or more control blocks in any order within the FORMAT section. Except for BEFORE GROUP OF and AFTER GROUP OF control blocks, each type of control block must be unique within the report.

EVERY ROW

The EVERY ROW keywords specify a default output format, including every input record that is passed to the report. If you use the EVERY ROW option, no other statements or control blocks are valid.

```
EVERY ROW ──────────────────────────────────────────────┤
```

Usage

This option formats the report in a simple default format, containing only the values that are passed to the REPORT program block through its arguments, and the names of the arguments.

You cannot modify the EVERY ROW statement with any of the statements listed in the "Statements in REPORT Control Blocks" section later in this chapter, nor can you include any control blocks in the FORMAT section. To display every record in a format other than the default format, use the ON EVERY ROW control block (as described in the "FORMAT Section Control Blocks" on page 6-27.)

The following example of a report definition uses the EVERY ROW option:

```
REPORT minimal(customer)
DEFINE customer RECORD LIKE customer.*
FORMAT
   EVERY ROW
END REPORT
```

Here is a portion of the output from the preceding default specification:

```
customer.customer_num  101
customer.fname         Ludwig
customer.lname         Pauli
customer.company       All Sports Supplies
customer.address1      213 Erstwild Court
customer.address2
customer.city          Sunnyvale
customer.state         CA
customer.zipcode       94086
customer.phone         408-789-8075

customer.customer_num  102
customer.fname         Carole
customer.lname         Sadler
customer.company       Sports Spot
customer.address1      785 Geary St
customer.address2
customer.city          San Francisco
customer.state         CA
customer.zipcode       94117
customer.phone         415-822-1289
```

Reports generated with the EVERY ROW option use as column headings the names of the variables that the report driver passes as arguments at run time. If all fields of each input records can fit horizontally on a single line, the default report prints the names across the top of each page, and the values beneath. Otherwise, it formats the report with the names down the left side of the page and the values to the right, as in the previous example. When a variable contains a NULL value, the default report prints only the name of the variable, with nothing for the value.

The following is another example of a brief report specification that uses the EVERY ROW statement. (Assume here that the cursor that retrieved the input records for this report was declared with an ORDER BY clause, so that no ORDER BY section is needed in this REPORT definition.)

```
DATABASE stores2

REPORT simple(order_num, customer_num, order_date)
DEFINE order_num LIKE orders.order_num,
       customer_num LIKE orders.customer_num,
       order_date LIKE orders.order_date
FORMAT
   EVERY ROW
END REPORT
```

The following is part of the output from the preceding REPORT definition.

```
order_num customer_num order_date

     1001          104  01/20/1993
     1002          101  06/01/1993
     1003          104  10/12/1993
     1004          106  04/12/1993
     1005          116  12/04/1993
     1006          112  09/19/1993
     1007          117  03/25/1993
     1008          110  11/17/1993
     1009          111  02/14/1993
     1010          115  05/29/1993
     1011          104  03/23/1993
     1012          117  06/05/1993
```

You can use the RIGHT MARGIN statement in the OUTPUT section to control the width of a report that uses the EVERY ROW statement.

FORMAT Section Control Blocks

Control blocks define the structure of a report, by specifying one or more statements to be executed when specific parts of the report are processed. If no data records are output to the report, then *none* of the statements in these blocks are executed. (See page 6-39.) Each of the seven types of control blocks is optional, but if you do not use the EVERY ROW keywords, you must include at least one control block in the FORMAT section.

Control Block	When Statements in Block Are Executed
FIRST PAGE HEADER	before processing of *the first page* begins
PAGE HEADER	before processing of the *each subsequent page* begins
BEFORE GROUP OF	before processing a *group* of sorted records
ON EVERY ROW	as *each record* is passed to the report
AFTER GROUP OF	after processing a *group* of sorted records
PAGE TRAILER	after processing of *each page* ends
ON LAST ROW	after the *last record* is passed to the report

A report can include BEFORE GROUP OF, AFTER GROUP OF, and ON EVERY ROW control blocks where the GROUP OF blocks reference the same variable. In this case, when the value of the variable changes, the report processes all BEFORE GROUP OF blocks before the ON EVERY ROW block, and the ON EVERY ROW block before all AFTER GROUP OF blocks.

The sequence in which the BEFORE GROUP OF and AFTER GROUP OF control blocks are executed depends upon the *sort list i*n the ORDER BY section. For example, assume that the ORDER BY section specifies a *sort list* of variables **a**, **b**, and **c** in that order (as illustrated on page 6-20). 4GL processes the control blocks in the following order, regardless of the physical sequence in which these control blocks appear within the FORMAT section:

```
BEFORE GROUP OF a                {1}
    BEFORE GROUP OF b            {2}
        BEFORE GROUP OF c        {3}
            ON EVERY ROW         {4}
        AFTER GROUP OF c         {3}
    AFTER GROUP OF b             {2}
AFTER GROUP OF a                 {1}
```

Figure 6-1 ***The Order of Group Processing, if "a,b,c" is the sort list in the ORDER BY Section***

In this example, a control block may be executed multiple times relative to any other block that is marked with a lower number in the right-hand column. Without an ORDER BY section, the default is the physical order of first mention of the variables in either BEFORE or AFTER GROUP OF control blocks.

Note: *New values assigned to variables in the PAGE HEADER control block are not available until after the first PRINT, SKIP, or NEED statement is executed in an ON EVERY ROW control block. This is done to guarantee that any group values printed in the PAGE HEADER control block have the same values as in the ON EVERY ROW control block. If a report written for release 4.0 or earlier of* INFORMIX-4GL *sets variables in the PAGE HEADER control block for use in the ON EVERY ROW control block, output from the report may be different when run with release 4.1 or later.*

AFTER GROUP OF

The AFTER GROUP OF control block specifies the action that 4GL takes after it processes a group of sorted records. Grouping is determined by the ORDER BY specification in the SELECT statement or in the REPORT definition.

statement is a report execution statement (page 6-39) or other 4GL statement.

variable is the name of a formal argument to the REPORT definition. You must pass at least the value of *variable* through the arguments.

Usage

A *group* of records is all of the input records that contain the same value for the variable whose name follows the AFTER GROUP OF keywords. This *group variable* must be passed through the report arguments. A report can include no more than one AFTER GROUP OF control block for any group variable.

When 4GL executes the statements in a AFTER GROUP OF control block, local variables have the values from the last record of the current group. From this perspective, the AFTER GROUP OF control block could be thought of as the "on last record of group" control block.

The Order of Processing AFTER GROUP OF Control Blocks

4GL executes the AFTER GROUP OF of control block on these occasions:

- Whenever the value of the group variable changes
- Whenever value of a higher-priority variable in the sort list changes.
- At the end of the report (after processing the last input record, but before 4GL executes any ON LAST ROW or PAGE TRAILER control blocks), each AFTER GROUP OF of control block is executed in ascending priority.

The section "The Sort List" on page 6-19 describes how input records are sorted according to a group variable (or a list of group variables), and the order of precedence among several variables in the sort list. How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If the records are already sorted, the AFTER GROUP OF block is executed after 4GL processes the last record of the group of records.
- If the records are not sorted, the AFTER GROUP OF control blocks may be

executed after any record, because the value of the group variable may change with each record. If the report includes no ORDER BY section (page 6-18), all AFTER GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section.

The AFTER GROUP OF control block is designed to work with sorted data. You can sort the records by specifying a sort list in either of the following:

- An ORDER BY section in the REPORT definition.
- The ORDER BY clause of the SELECT statement in the report driver.

To sort data in the REPORT definition (with an ORDER BY section), make sure that the name of the group variable appears in both the ORDER BY section and also in the AFTER GROUP OF control block.

To sort data in the ORDER BY clause of SELECT statement, do the following:

- Use the column name in the ORDER BY clause of the SELECT statement as the group variable in the AFTER GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure that you include an ORDER EXTERNAL BY section in the report to specify the precedence of variables in the sort list.

If you specify sort lists in both the report driver and in the report definition, the sort list in the ORDER BY section of the REPORT takes precedence. The sort list can include more than one variable. In this case, the report sorts the records by values in the first variable (highest priority). Records having the same value for the first variable are then ordered by the second variable, and so on, until records having the same values on all other variables are ordered by the last variable (lowest priority) in the sort list.

The GROUP Keyword in Aggregate Functions

In the AFTER GROUP OF control block, you can include the GROUP keyword to qualify aggregate report functions like AVG( ), SUM( ), MIN( ), or MAX( ):

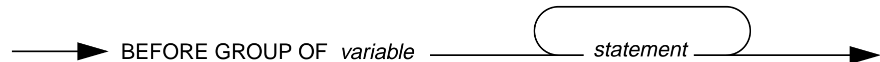
```
AFTER GROUP OF r.order_num
    PRINT "", r.order_date, 7 SPACES,
        r.order_num USING "###&",
        8 SPACES, r.ship_date, " ",
        GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"

AFTER GROUP OF r.customer_num
    PRINT 42 SPACES, "--------------------"
    PRINT 42 SPACES,
        GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

The GROUP keyword to qualify an aggregate function is only valid within the AFTER GROUP OF control block. It is not valid, for example, in the BEFORE GROUP OF control block: The aggregates report functions of 4GL are described on pages 6-46 and 4-13.

After the last input record is processed, 4GL executes the AFTER GROUP OF control blocks before it executes the ON LAST ROW control block.

BEFORE GROUP OF

The BEFORE GROUP OF control block specifies what action 4GL takes before it processes a group of input records. Group hierarchy is determined by the ORDER BY specification in the SELECT statement or in the REPORT definition.



statement is a report execution statement (page 6-39) or other 4GL statement.

variable is the name of a variable from the list of formal arguments to the REPORT definition. You must pass at least the value of *variable* through the arguments of the REPORT definition.

Usage

A *group* of records is all of the input records that contain the same value for the variable specified after the BEFORE GROUP OF keywords. You can include no more than one BEFORE GROUP OF control block for each group variable.

When 4GL executes the statements in a BEFORE GROUP OF control block, the report variables have the values from the first record of the new group. From this perspective, the BEFORE GROUP OF control block could be thought of as the "on first record of group" control block.

The Order of Processing BEFORE GROUP OF Control Blocks

Each BEFORE GROUP OF block is executed in highest-to-lowest priority order at the start of a report (after any FIRST PAGE HEADER or PAGE HEADER control blocks, but before processing the first record), and on these occasions:

- Whenever the value of the group variable changes (after any AFTER GROUP OF block for the old value completes execution).
- Whenever value of a higher-priority variable in the sort list changes (after any AFTER GROUP OF block for the old value completes execution).

How often the value of the group variable changes depends in part on whether the input records have been sorted by the SELECT statement:

- If the records are already sorted, the BEFORE GROUP OF block executes before 4GL processes the first record of the group of records.
- If the records are not sorted, the BEFORE GROUP OF control blocks may be executed after any record, because the value of the group variable may change with each record. If the report includes no ORDER BY section (6-18), all BEFORE GROUP OF control blocks are executed in the same order in which they appear in the FORMAT section.

The BEFORE GROUP OF control block is designed to work with sorted data. You can sort the records by specifying a sort list in either of the following:

- An ORDER BY section in the REPORT definition.
- The ORDER BY clause of the SELECT statement in the report driver.

To sort data in the REPORT definition (with an ORDER BY section), make sure that the name of the group variable appears in both the ORDER BY section and also in the BEFORE GROUP OF control block.

To sort data in the ORDER BY clause of SELECT statement, do the following:

- Use the column name in the ORDER BY clause of the SELECT statement as the group variable in the BEFORE GROUP OF control block.
- If the report contains BEFORE or AFTER GROUP OF control blocks, make sure that you include an ORDER EXTERNAL BY section in the report to specify the precedence of variables in the sort list.

If you specify sort lists in both the report driver and in the report definition, the sort list in the ORDER BY section of the REPORT takes precedence.

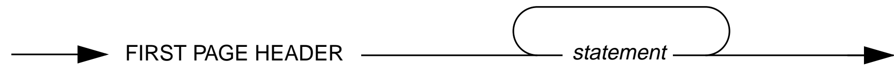
When 4GL starts to generate a report, it first executes the BEFORE GROUP OF control blocks in descending order of priority before it executes the ON EVERY ROW control block. (See also Figure 6-1 on page 6-27.)

If the report is not already at the top of the page, the SKIP TO TOP OF PAGE statement in a BEFORE GROUP OF control block causes the output for each group to start at the top of a page.

```
BEFORE GROUP OF r.customer_num
   SKIP TO TOP OF PAGE
```

FIRST PAGE HEADER

This control block specifies action that INFORMIX-4GL takes before it begins processing the first input record.You can use it, for example, to specify what (if any) information appears near the top of the first page of the report.



statement is a report execution statement (page 6-39) or other 4GL statement.

Usage

Because INFORMIX-4GL executes the FIRST PAGE HEADER control block *before* it generates any output, you can use this control block to initialize variables that you use in the FORMAT section. In the following example, from a report that produces multiple labels across the page, the FIRST PAGE HEADER does not display any information.

```
FIRST PAGE HEADER
   {Nothing is displayed in this control block.  It just
    initializes variables that are used in the ON EVERY ROW
    control block.}

   {Initialize label counter.}
   LET i = 1

   {Determine label width; allow 8 spaces between labels).}
   LET l_size = 72/count1

   {Divide 8 spaces among the labels across the page.}
   LET white = 8/count1
```

If a report driver includes START REPORT and FINISH REPORT statements, but no data records are passed to the report, this control block is not executed.

Displaying Titles and Headings

As its name implies, you can also use a FIRST PAGE HEADER control block to produce a title page, as well as column headings. On the first page of a report, this control block overrides any PAGE HEADER control block. That is, if both a FIRST PAGE HEADER and PAGE HEADER control block exist, output from the first appears at the beginning of the first page, and output from the second begins all subsequent pages.

The TOP MARGIN (set in the OUTPUT section) determines how close the header appears to the top of the page.

Restrictions on the List of Statements

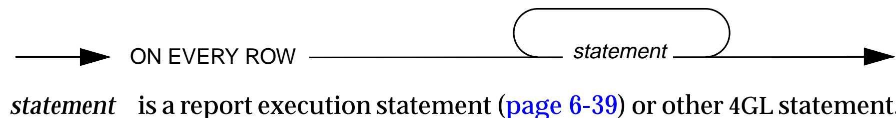
The following restrictions apply to FIRST PAGE HEADER control blocks:

- You cannot include a SKIP *integer* LINES statement inside a loop within this control block.
- The NEED statement is not valid within this control block.
- If you use an IF ... THEN ... ELSE statement within this control block, the number of lines displayed by any PRINT statements following the THEN keyword must be equal to the number of lines displayed by any PRINT statements following the ELSE keyword.
- If you use a CASE, FOR, or WHILE statement that contains a PRINT statement within this control block, you must terminate the PRINT statement with a semicolon (`;`) symbol. The semicolon suppresses any LINEFEED characters in the loop, keeping the number of lines in the header constant from page to page.
- You cannot use a PRINT *filename* statement to read and display text from a file within this control block.

Corresponding restrictions also apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in PAGE HEADER (page 6-37) and PAGE TRAILER (page 6-38) control blocks.

ON EVERY ROW

The ON EVERY ROW control block specifies the action to be taken by 4GL for every input record that is passed to the REPORT definition.



statement is a report execution statement (page 6-39) or other 4GL statement.

Usage

INFORMIX-4GL executes the statements within the ON EVERY ROW control block for each new input record that is passed to the report.

The following example is from a report that lists all the customers, their addresses, and their telephone numbers across the page:

```
ON EVERY ROW
  PRINT customer_num USING "###&",
     COLUMN 12, fname CLIPPED, 1 SPACE,
     lname CLIPPED, COLUMN 35, city CLIPPED,
     ", " , state,  COLUMN 57, zipcode,
     COLUMN 65, phone
```

The next example displays information about items and their prices:

```
ON EVERY ROW
  PRINT snum USING "##&", COLUMN 10, manu_code, COLUMN 18,
    description CLIPPED, COLUMN 38, quantity USING "##&",
    COLUMN 43, unit_price USING "$$$$.&&",
    COLUMN 55, total_price USING "$$,$$$,$$$.&&"
```

The next example is from a mailing label report:

```
ON EVERY ROW
  IF (city IS NOT NULL) AND
         (state IS NOT NULL) THEN
      PRINT fname CLIPPED, 1 SPACE, lname
      PRINT company
      PRINT address1
      IF (address2 IS NOT NULL) THEN PRINT address2
      PRINT city CLIPPED ", " , state, 2 SPACES, zipcode
      SKIP TO TOP OF PAGE
  END IF
```

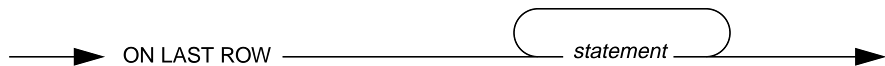
4GL delays processing the PAGE HEADER control block (or the FIRST PAGE HEADER control block, if it exists) until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW control block.

Group Control Blocks

If a BEFORE GROUP OF control block is triggered by a change in the value of a variable, 4GL executes all appropriate BEFORE GROUP OF control blocks (in the order of their priority) before it executes the ON EVERY ROW control block. Similarly, if execution of an AFTER GROUP OF control block is triggered by a change in the value of a variable, then 4GL executes all appropriate AFTER GROUP OF control blocks (in the reverse order of their priority) before it executes the ON EVERY ROW control block.

ON LAST ROW

The ON LAST ROW control block specifies the action that 4GL is to take after it processes the last input record that was passed to the REPORT definition and encounters the FINISH REPORT statement.



statement is a report execution statement (page 6-39) or other 4GL statement.

Usage

The statements in the ON LAST ROW control block are executed after the statements in the ON EVERY ROW and AFTER GROUP OF control blocks, if these are present.

When **4GL** processes the statements in an ON LAST ROW control block, the variables that the report is processing still have the values from the final record that the report processed. The ON LAST ROW control block can use aggregate functions to display report totals, as in this example:

```
ON LAST ROW
  SKIP 1 LINE
  PRINT COLUMN 12, "TOTAL NUMBER OF CUSTOMERS:",
        COLUMN 57, COUNT(*) USING "#&"
```

PAGE HEADER

The PAGE HEADER control block specifies the action that 4GL takes before it begins processing each page of the report. It can specify what information, if any, appears at the top of each new page of the report.

statement is a report execution statement (page 6-39) or other 4GL statement.

Usage

You can use a PAGE HEADER control block to display column headings. The next example produces column headings for printing data across the page.

```
PAGE HEADER
  PRINT "NUMBER",
  COLUMN 12, "NAME",
  COLUMN 35, "LOCATION",
  COLUMN 57, "ZIP",
  COLUMN 65, "PHONE"
  SKIP 1 LINE
```

This control block is executed whenever a new page is added to the report. The TOP MARGIN specification (in the OUTPUT section) affects how many blank lines appear above the output produced by *statements* in the PAGE HEADER control block. You can use the PAGENO operator in a PRINT statement within a PAGE HEADER control block to display the current page number automatically at the top of every page.

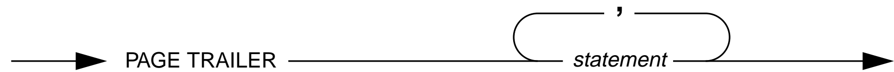
The FIRST PAGE HEADER control block overrides this control block on the *first* page of a report.

New group values can appear in the PAGE HEADER control block when this control block is executed after a simultaneous end-of-group and end-of-page situation. 4GL delays the processing of the PAGE HEADER control block until it encounters the first PRINT, SKIP, or NEED statement in the ON EVERY ROW, BEFORE GROUP OF, or AFTER GROUP OF control block. This guarantees that any group columns printed in the PAGE HEADER control block have the same values as the columns printed in the ON EVERY ROW control block.

The same restrictions apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in the PAGE HEADER control block as apply to the FIRST PAGE HEADER (page 6-34) and PAGE TRAILER (page 6-38) control blocks.

PAGE TRAILER

The PAGE TRAILER control block specifies what information, if any, appears at the bottom of each page of the report.



statement is a report execution statement (page 6-39) or other 4GL statement.

Usage

4GL executes the *statements* in the PAGE TRAILER control block before the PAGE HEADER control block when a new page is needed. New pages can be initiated by any of the following conditions:

- PRINT attempts to print on a page that is already full.
- SKIP TO TOP OF PAGE is executed.
- SKIP *n* LINES specifies more lines than are available on the current page.
- NEED specifies more lines than are available on the current page.

You can use the PAGENO operator in a PRINT statement within a PAGE TRAILER control block to display the page number automatically at the bottom of every page, as in the following example:

```
PAGE TRAILER
   PRINT COLUMN 28,
      PAGENO USING "page <<<<"
```

The BOTTOM MARGIN specification (in the OUTPUT section) affects how close to the bottom of the page the output displays the page trailer.

Restrictions on the List of Statements

The number of lines produced by the PAGE TRAILER control block cannot vary from page to page, and must be unambiguously expressed. See the list of specific restrictions that apply to CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements in the FIRST PAGE HEADER (page 6-34) control block.

Statements in REPORT Control Blocks

The control blocks determine *when* 4GL takes an action in a report; within each control block, the statements determine *what action* 4GL takes. The list of statements within a control block terminates when another control block begins, or else when the END REPORT keywords that terminate the REPORT program block are encountered. You can include most 4GL statements in a control block, as well as several statements that can be used only in the FORMAT section of a REPORT definition.

SQL Statements

The REPORT definition cannot include SQL statements. You can include any SQL statements that the report requires in the report driver (page 6-5), rather than in the REPORT program block.

Other INFORMIX-4GL Statements

The 4GL statements most frequently used in the control blocks of reports are CASE, FOR, IF, LET, and WHILE. These statements have the same syntax that they have elsewhere in 4GL applications, as Chapter 3 describes. (Remember that any local variables referenced in such statements must be declared in the DEFINE section of the REPORT definition.)

Statements Valid Only in the FORMAT Section

Four statements (sometimes called *report execution statements*), can appear only in control blocks of the FORMAT section of a REPORT definition:

Statement	Effect
NEED	Forces a page break, unless some specified number of lines are available on the current page of the report.
PAUSE	Allows the user to control scrolling of screen output. (This statement has no effect if output is sent to any destination except the screen.)
PRINT	Appends a specified item to the output of the report.
SKIP	Inserts blank lines into a report, or forces a page break.

Descriptions of these report execution statements follow.

NEED

This statement causes any subsequent display to start on the next page, if fewer than the specified number of lines remain between the current line and the bottom margin of the current page of report output.

```
NEED lines LINES ────────────────────────────────────────────┤
```

lines is an integer expression (page 3-338) that returns a positive whole number.

Usage

This statement has the effect of a conditional SKIP TO TOP OF PAGE statement, the "condition" being that the number to which the integer expression evaluates is greater than the number of lines that remain on the current page.

The NEED statement can prevent the report from dividing parts of the output that you want to keep together on a single page. In the following example, the NEED statement causes the PRINT statement to send output to the next page, unless at lease six lines remain on the current page:

```
AFTER GROUP OF r.order_num
    NEED 6 LINES
    PRINT " ",r.order_date, 7 SPACES,
        GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
```

The *lines* value specifies how many lines must remain between the line above the current character position and the bottom margin for the next PRINT statement to produce output on the current page. If the number of remaining lines on the page is less than *lines*, then 4GL prints both the PAGE TRAILER and the PAGE HEADER.

The NEED statement does not include the BOTTOM MARGIN value when it compares *lines* to the number of lines remaining on the current page.

The NEED statement is not valid in FIRST PAGE HEADER, PAGE HEADER, nor PAGE TRAILER control blocks.

References

PAUSE, PRINT, REPORT, SKIP

PAUSE

The PAUSE statement temporarily suspends output to the screen, until the user presses RETURN.

PAUSE

string is a quoted string that PAUSE displays. If you do not supply a message, PAUSE displays no message.

Usage

Output is sent by default to the screen unless the START REPORT statement or the OUTPUT section specifies a destination for report output. The PAUSE statement can be executed only if the report sends its output to the screen. It has no effect if you include a TO clause in either of these contexts:

- In the OUTPUT section of the REPORT definition (page 6-13).
- In the START REPORT statement of the report driver (page 3-271).

Include the PAUSE statement in the PAGE HEADER or PAGE TRAILER block of the report. For example, the following code causes **INFORMIX-4GL** to skip a line and pause at the end of each page of report output displayed on the screen.

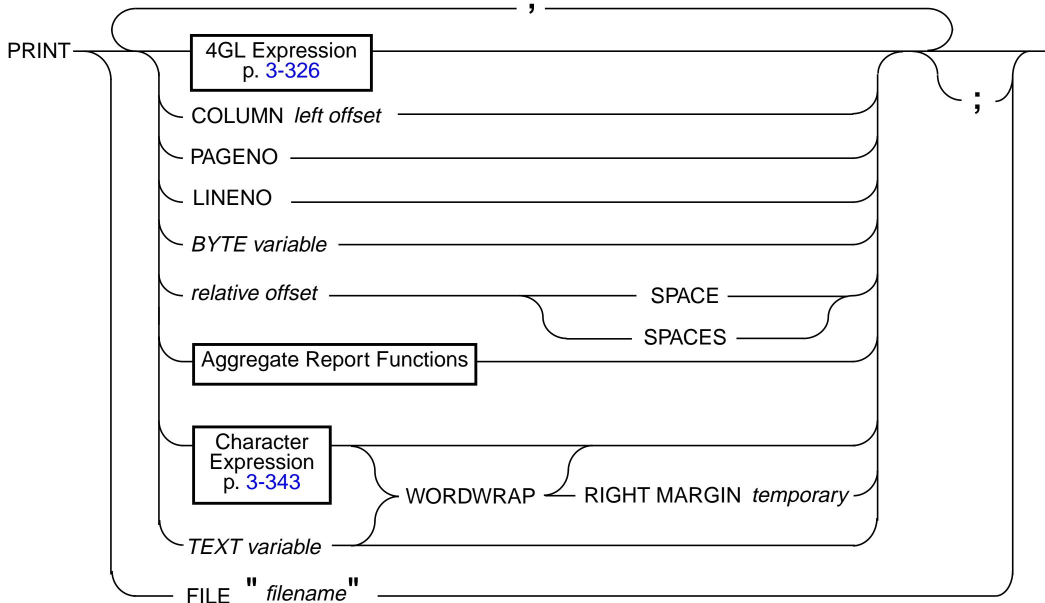
```
PAGE TRAILER
    SKIP 1 LINE
    PAUSE "Press RETURN to display next screen."
```

References

NEED, PRINT, REPORT, SKIP

PRINT

The PRINT statement produces output from a REPORT definition.



BYTE variable is the identifier of a 4GL variable of data type BYTE.

filename is a character string, enclosed between quotation (") marks, and specifying the name of an ASCII file to include in the output from the report. The *filename* can include a pathname.

left offset is an expression that evaluates to a positive whole number, specifying a character position offset (from the *left margin*) no greater than the difference (*right margin – left margin)*.

relative offset is an expression that evaluates to a positive whole number, specifying an offset (from the *current character position*) no greater than the difference (*right margin – current position)*.

temporary is an expression that evaluates to a positive whole number, specifying the *absolute* position of a *temporary right margin*.

TEXT variable is the identifier of an 4GL variable of the TEXT data type.

See page 3-338 for the syntax of 4GL expressions like the *left offset*, *relative offset*, and *temporary* terms that evaluate to integer values.

Usage

This statement can include character data in the form of an ASCII *file*, a TEXT variable, or a comma-separated *expression list* of character expressions in the output of the report. (For a *TEXT variable* or *filename*, you cannot specify additional output in the same PRINT statement.) You cannot display a BYTE value. Unless its scope of reference is global or the current module, any program variable in *expression list* must be declared in the DEFINE section.

Output is sent to the destination specified in the REPORT TO clause of the OUTPUT section, or in the TO clause of the START REPORT statement of the report driver. Otherwise, the screen (page 6-14) is the destination.

NLS

When NLS is active, the settings in the NLS environment variables LC_MONETARY and LC_NUMERIC affect the default numeric and monetary formatting. For example, the number 2345.67 in a US English locale prints as 2,345.67. With LC_NUMERIC set for the French locale, where the thousands separator is the comma and the decimal separator the period, the value prints as 2.345,67. Note that setting either DBFORMAT or DBMONEY overrides any settings in LC_NUMERIC or LC_MONETARY.

The following example is from the FORMAT section of a REPORT definition that displays both quoted strings and values from rows of the **customer** table:

```
FIRST PAGE HEADER
   PRINT COLUMN 30, "CUSTOMER LIST"
   SKIP 2 LINES
   PRINT "Listings for the State of ", thisstate
   SKIP 2 LINES
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
PAGE HEADER
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
ON EVERY ROW
   PRINT customer_num USING "###&", COLUMN 12, fname CLIPPED,
      1 SPACE, lname CLIPPED, COLUMN 35, city CLIPPED, ", " ,
      state, COLUMN 57, zipcode, COLUMN 65, phone
```

The FILE Option

The PRINT FILE statement reads the contents of the specified *filename* into the report, beginning at the current character position. This permits you to insert a multiple-line character string into the output of a report. The following example uses the PRINT FILE statement to include the body of a form letter from file **occupant.let** in the output of a report that generates letters:

```
PRINT "Dear", 1 SPACES, "fname",","
PRINT FILE "/usr/claire/occupant.let"
```

If *filename* stores the value of a TEXT variable, then the PRINT FILE "*filename*" statement has the same effect as specifying PRINT *variable*. (But only PRINT *variable* can include the WORDWRAP operator, as described on page 6-50.)

The Character Position

PRINT statement output begins at the *current character position*, sometimes called simply the "current position." On each page of a report, the initial default character position is the first character position in the first line. This can be offset horizontally and vertically by *margin* and *header* specifications, and by executing any of the following statements:

- The SKIP statement moves it down to the left margin of a new line.
- The NEED statement can conditionally move it to a new page.
- The PRINT statement moves it horizontally (and sometimes down).

Unless you use the keyword CLIPPED or USING, values are displayed with widths (including any sign) that depend on their declared data types:

Data Type	Default Display Width (in *characters*)
BYTE	12. (**4GL** displays the string "`<byte value>`" as the only output.)
CHAR	The length from the data-type declaration.
DATE	10.
DATETIME	From 2 to 25, as implied in the data-type declaration.
DECIMAL	(2 + *m*), for *m* the precision from the data-type declaration.
FLOAT	14.
INTEGER	11.
INTERVAL	From 3 to 25, as implied in the data-type declaration.
MONEY	(3 + *m*), for *m* the precision from the data-type declaration.
SMALLFLOAT	14.
SMALLINT	6.
TEXT	The data length of the character string.
VARCHAR	The data length of the character string.

Unless you specify the FILE or WORDWRAP option, the PRINT statement displays its output on a single line. This fragment displays output on two lines:

```
PRINT fname, lname
PRINT city, ", " , state, 2 SPACES, zipcode
```

If you terminate a PRINT statement with a semicolon (;) symbol, however, you suppress the implicit LINEFEED character at the end of the line. The next example has the same effect as the PRINT statements in the previous example:

```
PRINT fname;
PRINT lname
PRINT city, ", ";
PRINT state, 2 SPACES, zipcode
```

The Expression List

The *expression list* of a PRINT statement returns one or more values that can be displayed as printable characters. The following built-in functions and operators can appear in the expression list of PRINT. Some of these can appear *only* in a REPORT program block. Letter superscripts indicate restrictions on the context where some of the following can appear within an 4GL program. (All of these built-in functions and operators are described in Chapter 4.)

ASCII	DATE( )	MIN( )s	TIME
AVG( )[s]	DAY( )	MDY( )	TODAY
CLIPPED	EXTEND( )	MONTH( )	UNITS
COLUMN	GROUP[r]	PAGENO[r]	USING
COUNT(*)[s]	LENGTH( )	PERCENT(*)[r]	WEEKDAY( )
CURRENT	LINENO[r]	SPACES[r]	YEAR
DATE	MAX( )[s]	SUM( )[s]	YEAR( )

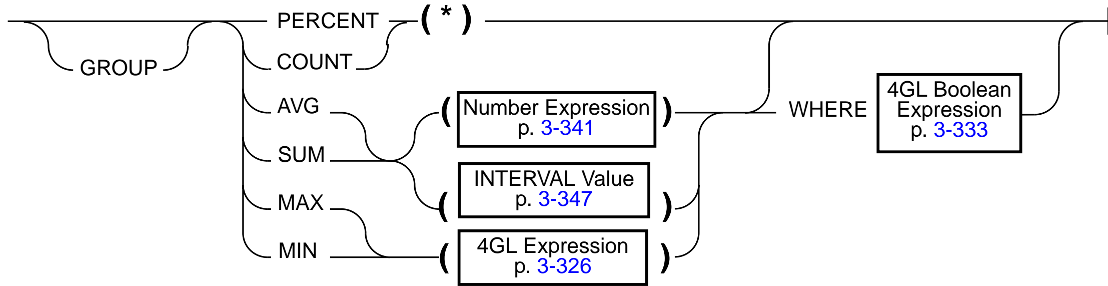
r *You can use these expressions only within the FORMAT section of a REPORT definition. (A description appears later in this section.)*

s *You can use these aggregate functions only in the FORMAT section of a REPORT, or in statements of SQL. (Note that the PERCENT(*) aggregate cannot appear in these SQL statement.)*

If the expression list applies the USING operator to format a DATE or MONEY value, the format-string of the USING operator (page 4-91) takes precedence over the DBDATE, DBMONEY, or DBFORMAT environment variables.

Aggregate Report Functions

Aggregate report functions summarize data from several records in a report. The syntax and effects of aggregates in a report resemble those of SQL aggregate functions, but are *not* identical. (See the *Informix Guide to SQL: Reference* for the syntax of SQL aggregate functions in SQL statements.)



The *expression* (in parentheses) that SUM( ), AVG( ), MIN( ), or MAX( ) takes as an argument is typically of a *number* or INTERVAL data type; ARRAY, BYTE, RECORD, or TEXT are not valid. The AVG( ), SUM( ), MIN( ), and MAX( ) aggregates ignore input records for which their arguments have NULL values, but each returns NULL if every record has a NULL value for the argument.

The GROUP Keyword

This optional keyword causes the aggregate function to include data only for a *group* of records that have the same value on a variable that you specify in an AFTER GROUP OF control block. An aggregate function can only include the GROUP keyword within an AFTER GROUP OF control block.

The WHERE Clause

The optional WHERE clause allows you to select among records passed to the report, so that only records for which the Boolean expression is TRUE are included. (See the section "4GL Boolean Expressions" on page 3-333.)

The AVG( ) and SUM( ) Aggregates

These evaluate as the *average* (that is, the *arithmetic mean* value) and the total (respectively) of *expression* among all records, or among records qualified by the optional WHERE clause and any GROUP specification.

The COUNT (*) and PERCENT (*) Aggregates

These return, respectively, the *total number* of records qualified by the optional WHERE clause, and the *percentage* of the total number of records in the report. You must include the (*) symbols.

The following fragment of a REPORT definition uses the AFTER GROUP OF control block and GROUP keyword to form sets of records according to how many items are in each order. The last PRINT statement calculates the total price of each order, then adds a shipping charge, and prints the result.

```
AFTER GROUP OF number
  SKIP 1 LINE
  PRINT 4 SPACES, "Shipping charges for the order:  ",
    ship_charge USING "$$$$.&&"
  PRINT 4 SPACES,  "Count of small orders:  ",
    count(*) WHERE total_price < 200.00 USING "##,###"
  SKIP 1 LINE
  PRINT 5 SPACES, "Total amount for the order:  ",
    ship_charge + GROUP SUM(total_price) USING "$$,$$$,$$$.&&"
```

Since no WHERE clause is specified, GROUP SUM( ) combines the **total_price** of *every* item in the group comprising the order.

The MIN( ) and MAX( ) Aggregates

These evaluates as the *minimum* value and *maximum* value (respectively) for *expression* among all records, or among records qualified by the optional WHERE clause and any GROUP specification. For character data, *greater than* means "after" in the ASCII collating sequence, where a> A> 1, and *less than* means "before" in the ASCII sequence, where 1< A< a. For DATETIME or DATE data, *greater than* means "later" and *less than* means "earlier" in time. Appendix G lists the ASCII collating sequence.

The ASCII Operator

This returns the ASCII character whose numeric code you specify, just as described in Chapter 4, with one exception: To print a NULL character in a report, call the ASCII operator with 0 in a PRINT statement. For example, the following statement prints the NULL character:

PRINT ASCII 0

ASCII 0 only displays a NULL character in the PRINT statement. In other contexts, ASCII 0 returns a blank space. (See also the description of ASCII on page 4-28.)

The COLUMN Operator

The COLUMN operator can appear in PRINT statements to move the character position forward within the current line. It has the following syntax:

COLUMN *left offset*

left offset is an integer expression (page 3-338) that specifies a character position offset (from the *left margin*) no greater than the difference (*right margin* – *left margin)*.

This moves the character position to the specified *left offset*, where `1` is the first position after the left margin. If *current position* is greater than *left offset*, the specification is ignored. (See also the description of COLUMN on page 4-40).

The LINENO Operator

This returns the value of the *line number* of the report line that is currently printing. INFORMIX-4GL computes the line number by calculating the number of lines from the top of the current page, including the TOP MARGIN. In the following example, a PRINT statement instructs INFORMIX-4GL to calculate and display the current line number, beginning in the tenth character position after the left margin.

```
IF (LINENO > 9) THEN
   PRINT COLUMN 10, LINENO USING "Line <<<"
END IF
```

The PAGENO Operator

This returns the value of the number of the page that INFORMIX-4GL is currently printing. The next example conditionally prints the value of PAGENO, using the USING operator to format it, if its value is less than 10,000.

```
IF (PAGENO < 10000) THEN
   PRINT COLUMN 28, PAGENO USING "page <<<<"
END IF
```

You can use PAGENO in the PAGE HEADER or PAGE TRAILER, or in other control blocks, to number sequentially the pages of a report.

If you use the SQL aggregate COUNT(*) in the SELECT statement to find how many records are returned by the query, and if the number of records that appear on each page of output is both fixed and known, you can calculate the total number of pages, as in the following example:

```
SELECT COUNT(*) num  FROM customer INTO TEMP cnt
SELECT * FROM customer, cnt   --Note temp table in FROM clause
                                    --and no join is necessary
    . . .
FORMAT
    FIRST PAGE HEADER
    LET y = cnt/50  --assumes 50 records per page; you must
                    --round up if there is a remainder.}
    PAGE TRAILER
    PRINT "Page ", PAGENO USING "<<" " of ", y USING "<<"
```

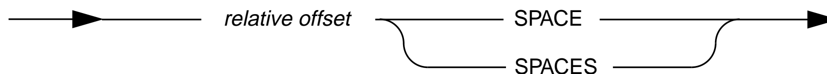
If the calculated number of pages were 20, the first page trailer would be:

```
Page 1 of 20
```

PAGENO is incremented with each page, so the last page trailer would be:

```
Page 20 of 20
```

The SPACE or SPACES Operator

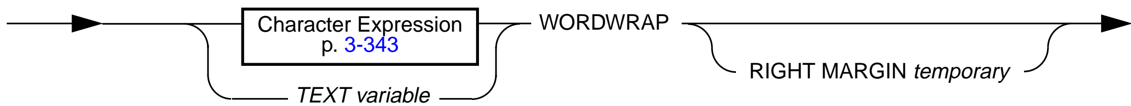


relative offset is an integer expression (page 3-338) that returns a positive number, specifying an offset (from the *current character position*) no greater than the difference (*right margin* – *current position)*.

This returns a string of blanks, equivalent to a quoted string containing the specified number of blanks. Outside PRINT statements, SPACE (or SPACES) and its operand must appear in parentheses. The following statements use this operator to separate values in PRINT statements, to concatenate 6 blank spaces to the string "=ZIP," and to print the result after the variable **zipcode**:

```
FORMAT
  ON EVERY ROW
     LET mystring = (6 SPACES), "=ZIP"
     PRINT fname, 2 SPACES, lname
     PRINT company
     PRINT address1
     PRINT city, ", " , state, 2 SPACES, zipcode, mystring
```

The WORDWRAP Operator



temporary is a literal integer (page 3-340) specifying a character position (from left edge of the page) of a temporary right margin.

TEXT variable is the name of a 4GL variable of the TEXT data type.

The WORDWRAP operator automatically wraps successive segments of long character strings onto successive lines of report output. Any string value that is too long to fit between the current position and the right margin is divided into segments and displayed between temporary margins:

- The current character position becomes the temporary left margin.
- Unless you specify RIGHT MARGIN *temporary*, the right margin defaults to 132, or to the *size* from the RIGHT MARGIN clause of the OUTPUT section.

Specify WORDWRAP RIGHT MARGIN *temporary* to set a temporary right margin, counting from the left edge of the page. This cannot be smaller than

the current character position, nor greater than 132 (or than the *size* from the RIGHT MARGIN clause of the OUTPUT section). The current character position becomes the temporary left margin. These temporary values override the specified or default left and right margins from the OUTPUT section.

After the PRINT statement has executed, any explicit or default margins from the OUTPUT section are restored.

The following PRINT statement specifies a temporary left margin in column 10 and a temporary right margin in column 70 to display the character string that is stored in the 4GL variable called **mynovel**:

```
PRINT COLUMN 10, mynovel WORDWRAP RIGHT MARGIN 70
```

Tabs, Line Breaks, and Page Breaks with WORDWRAP

The data string can include printable ASCII characters. It can also include the TAB (ASCII 9), LINEFEED (ASCII 10), and ENTER (ASCII 13) characters to partition the string into "words" consisting of substrings of other printable characters. Other non-printable characters may cause runtime errors. If the data string cannot fit between the margins of the current line, 4GL breaks the line at a *word* division, padding line with blanks at the right.

From left to right, 4GL expands any TAB character to enough blank spaces to reach the next TAB stop. By default, TAB stops are in every eighth column, beginning at the left-hand edge of the page. If the next TAB stop or a string of blank characters extends beyond the right margin, 4GL takes these actions:

- Prints blank characters only to the right margin.
- Discards any remaining blank characters from the blank string or TAB.
- Starts a new line at the temporary left margin.
- And processes the next word.

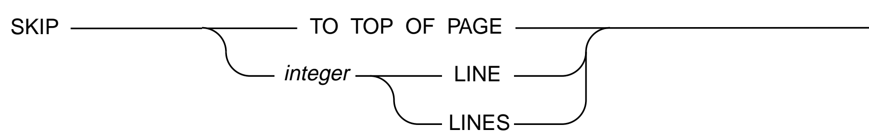
4GL starts a new line when a word plus the next blank space cannot fit on the current line. If all words are separated by a single space, this creates an even left margin. 4GL applies the following rules (in descending order of precedence) to the portion of the data string within the right margin:

- Break at any LINEFEED, or ENTER, or LINEFEED, ENTER pair.
- Break at the last blank (ASCII 32) or TAB character before the right margin.
- Break at the right margin, if no character farther to the left is a blank, ENTER, TAB, or LINEFEED character.

4GL maintains page discipline under the WORDWRAP option. If the string is too long for the current page, 4GL executes the statements in any page trailer and header control blocks before continuing output onto a new page.

SKIP

The SKIP statement can insert a specified number of blank lines into the output of a REPORT definition, or else advance the character position to the top of the next page of report output.



integer is a literal integer (page 3-340) that specifies the number of lines.

Usage

The SKIP statement allows you to insert blank lines into report output, or to skip to the top of the next page, as if you had included an equivalent number of PRINT statements without specifying any expression list. The LINE and LINES keywords are synonyms in the SKIP statement.

Output from any PAGE HEADER, or PAGE TRAILER control block appears in its usual location. This program fragment prints names and addresses:

```
FIRST PAGE HEADER
   PRINT COLUMN 30, "CUSTOMER LIST"
   SKIP 2 LINES
   PRINT "Listings for the State of ", thisstate
   SKIP 2 LINES
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
PAGE HEADER
   PRINT "NUMBER", COLUMN 12, "NAME", COLUMN 35, "LOCATION",
      COLUMN 57, "ZIP", COLUMN 65, "PHONE"
   SKIP 1 LINE
ON EVERY ROW
   PRINT customer_num USING "####",
   COLUMN 12, fname CLIPPED, 1 SPACE,
      lname CLIPPED, COLUMN 35, city CLIPPED, ", " , state,
      COLUMN 57, zipcode, COLUMN 65, phone
```

Restrictions on SKIP Statements

The SKIP LINES statement cannot appear within a CASE statement, a FOR loop, nor a WHILE loop. The SKIP TO TOP OF PAGE statement cannot appear in a FIRST PAGE HEADER, PAGE HEADER, nor PAGE TRAILER control block.

Appendix

A

The Demonstration Database and Application

The stores demonstration database contains a set of tables that describe an imaginary business. You can access the data in the stores demonstration database by the demonstration programs that appear in this book, as well as by application programs that are listed in the documentation of other Informix products. The stores demonstration database is not MODE ANSI.

This appendix contains the following sections:

- Instructions for copying the demonstration database and application.
- A description of the structure of the tables in the stores demonstration database. For each table, the name and the data type of each column are listed. Any indexes on individual columns or on multiple columns are identified and classified as unique or as allowing duplicate values.
- A graphic map of the tables in the stores demonstration database, showing potential join columns.
- A discussion of the join columns that link some of the tables in the stores demonstration database, and illustrates how you can use these relationships to obtain information from multiple tables.
- A listing of the data contained in each table of the stores demonstration database.

- The form specifications, **INFORMIX-4GL** source code modules, and help message source code for the stores demonstration application.

Making a Copy of the Demonstration Database

To make a copy of the stores demonstration database, select a directory to store the copy (often your home directory) and make it your current working directory. At the system prompt, enter one of the following commands:

`i4gldemo` (for the **INFORMIX-4GL C Compiler Version**)
`r4gldemo` (for the **INFORMIX-4GL Rapid Development System**).

The program creates a subdirectory called **stores.dbs** in your current directory and places the stores demonstration database files there. It also copies all the demonstration programs, forms, and help files into the current directory. If you list the contents of your current directory, you will see filenames similar to these:

c_menu2.4gl	ch7qry2.4gl	ordmenu.4gl
ch10def.4gl	ch7query.4gl	report1.4gl
ch10def2.4gl	ch7upd.4gl	report2.4gl
ch10key.4gl	cust.per	report3.4gl
ch10key2.4gl	custcur.per	report4.4gl
ch10notf.4gl	custhelp.ex	report5.4gl
ch10when.4gl	custhelp.msg	report6.4gl
ch12cust.4gl	custmenu.4gl	report7.4gl
ch12ord.4gl	customer.per	stock1.per
ch7add.4gl	ordcur.per	wind1.4gl
ch7add2.4gl	order.per	wind2.4gl
ch7del.4gl		

Restoring the Demonstration Database

As you work with your copy of the stores demonstration database, you may change it in such a way that the illustrations in this manual no longer reflect what you actually see on your screen. This can happen if you enter new information into the demonstration database, delete the information that came with the database, or alter the structure of the tables, forms, or reports.

You can restore the database to its original condition (upon which the examples are based) by recreating the database by entering one of the following commands:

`i4gldemo` (for the **INFORMIX-4GL C Compiler Version**)
`r4gldemo` (for the **INFORMIX-4GL Rapid Development System**).

If your **INFORMIX-4GL** software has been installed according to the instructions provided in your installation letter, the files that make up the stores demonstration database are protected, so that you cannot make any changes to the original copy of the database.

Structure of the Tables

The stores demonstration database contains information about a fictitious sporting goods distributor that services stores in the Western United States. This database includes the following tables:

- **customer**
- **orders**
- **items**
- **stock**
- **catalog**
- **cust_calls**
- **manufact**
- **state**

The customer Table

The **customer** table contains information about the retail stores that place orders from the distributor. The columns of the **customer** table are as follows:

Column Name	Data Type	Description
customer_num	SERIAL(101)	system-generated customer number
fname	CHAR(15)	first name of store's representative
lname	CHAR(15)	last name of store's representative
company	CHAR(20)	name of store
address1	CHAR(20)	first line of store's address
address2	CHAR(20)	second line of store's address
city	CHAR(15)	city
state	CHAR(2)	state
zipcode	CHAR(5)	zip code
phone	CHAR(18)	phone number

The **customer_num** column is indexed and must contain unique values. The **zipcode** and **state** columns are indexed to allow duplicate values.

The orders Table

The **orders** table contains information about orders placed by the distributor's customers. The columns of the orders table are as follows:

Column Name	Data Type	Description
order_num	SERIAL(1001)	system-generated order number
order_date	DATE	date order entered
customer_num	INTEGER	customer number (from customer table)
ship_instruct	CHAR(40)	special shipping instructions
backlog	CHAR(1)	indicates order cannot be filled because the item is backlogged: y = yes n = no
po_num	CHAR(10)	customer purchase order number
ship_date	DATE	shipping date
ship_weight	DECIMAL(8,2)	shipping weight
ship_charge	MONEY(6)	shipping charge
paid_date	DATE	date order paid

The **order_num** column is indexed and must contain unique values. The **customer_num** column is indexed to allow duplicate values.

The items Table

An order can include one or more items. There is one row in the items table for each item in an order. The columns of the **items** table are as follows:

Column Name	Data Type	Description
item_num	SMALLINT	sequentially assigned item number for an order
order_num	INTEGER	order number (from orders table)
stock_num	SMALLINT	stock number for item (from stock table)
manu_code	CHAR(3)	manufacturer's code for item ordered (from manufact table)
quantity	SMALLINT	quantity ordered
total_price	MONEY(8,2)	quantity ordered × unit price = total price of item

The **order_num** column is indexed and allows duplicate values. A multiple-column index for the **stock_num** and **manu_code** columns also permits duplicate values.

The stock Table

The distributor carries 41 different types of sporting goods from various manufacturers. More than one manufacturer can supply a sporting good. For example, the distributor offers racer goggles from two manufacturers and running shoes from six manufacturers.

The **stock** table is a catalog of the items sold by the distributor. The columns of the **stock** table are as follows:

Column Name	Data Type	Description
stock_num	SMALLINT	stock number that identifies type of item
manu_code	CHAR(3)	manufacturer's code (from manufact table)
description	CHAR(15)	description of item
unit_price	MONEY(6,2)	unit price
unit	CHAR(4)	unit by which item is ordered: each pair case box
unit_descr	CHAR(15)	description of unit

The **stock_num** and **manu_code** columns are indexed and allow duplicate values. A multiple-column index for both the **stock_num** and **manu_code** columns allows only unique values.

The catalog Table

The **catalog** table describes each of the items in stock. Retail stores use this catalog when placing orders with the distributor. The columns of the **catalog** table are as follows:

Column Name	Data Type	Description
catalog_num	SERIAL(10001)	system-generated catalog number
stock_num	SMALLINT	distributor's stock number (from stock table)
manu_code	CHAR(3)	manufacturer's code (from manufact table)
cat_descr	TEXT	description of item
cat_picture	BYTE	picture of item (binary data)
cat_advert	VARCHAR(255, 65)	tag line underneath picture

The **catalog_num** column is indexed and must contain unique values. The **stock_num** and **manu_code** columns allow duplicate values. A multiple-column index for the **stock_num** and **manu_code** columns allows only unique values.

The **catalog** table appears only if you are using an **INFORMIX-OnLine** database engine.

The cust_calls Table

All customer calls for information on orders, shipments, or complaints are logged. The **cust_calls** table contains information about these types of customer calls. The columns of the **cust_calls** table are as follows:

Column Name	Data Type	Description
customer_num	INTEGER	customer number (from customer table)
call_dtime	DATETIME YEAR TO MINUTE	date and time call received
user_id	CHAR(18)	name of person logging call
call_code	CHAR(1)	type of call: B = billing error D = damaged goods I = incorrect merchandise sent L = late shipment O = other
call_descr	CHAR(240)	description of call
res_dtime	DATETIME YEAR TO MINUTE	date and time call resolved
res_descr	CHAR(240)	description of how call was resolved

A multiple-column index for both the **customer_num** and **call_dtime** columns allows only unique values. The **customer_num** column also has an index that allows duplicate values.

The manufact Table

Information about the nine manufacturers whose sporting goods are handled by the distributor is stored in the **manufact** table. The columns of the **manufact** table are as follows:

Column Name	Data Type	Description
manu_code	CHAR(3)	manufacturer's code
manu_name	CHAR(15)	name of manufacturer
lead_time	INTERVAL DAY(3) TO DAY	lead time for shipment of orders

The **manu_code** column has an index that requires unique values.

The state Table

The **state** table contains the names and postal abbreviations for the 50 states of the United States. It includes the following two columns:

Column Name	Data Type	Description
code	CHAR(2)	state code
sname	CHAR(15)	state name

The **code** column is indexed as unique.

Map of the Demonstration Database

Figure A-1 displays the column names of the tables in the demonstration database. Shading connecting a column in one table to a column in another table indicates columns that contain the same information.

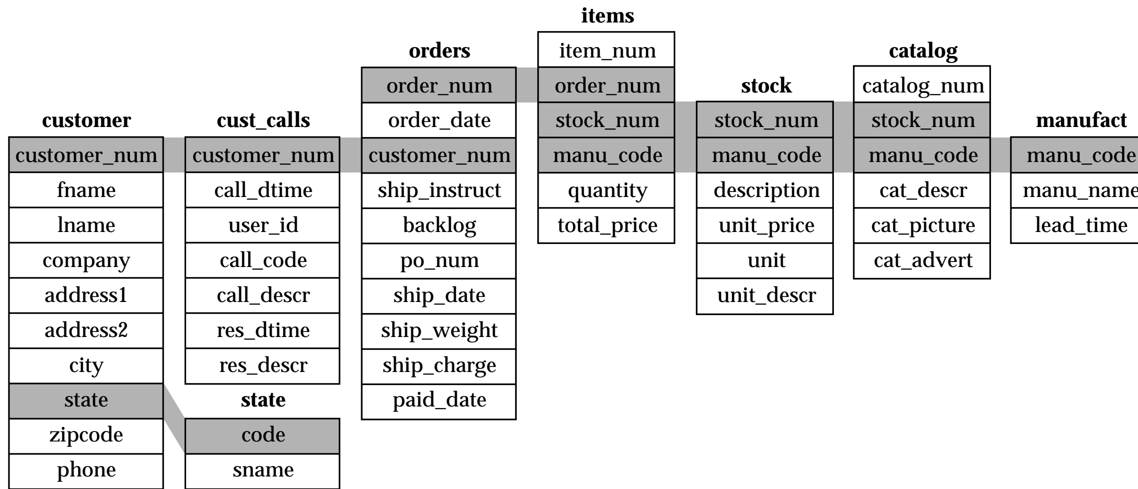


Figure A-1 Tables in the demonstration database

Join Columns that Link the Database

The tables of the demonstration database are linked together by the *join columns* shown in Figure A-1 and identified in this section. You can use these columns to retrieve and display information from several tables at once, as if the information had been stored in a single table. Figure A-1 through Figure A-8 show the join relationships among tables, and how information stored in one table supplements information stored in others.

Join Columns in the customer and orders Tables

The **customer_num** column joins the **customer** table and the **orders** table, as shown in Figure A-2.

customer_num	fname	lname	
101	Ludwig	Pauli	***customer*** *table (detail)*
102	Carole	Sadler	
103	Philip	Currie	
104	Anthony	Higgins	

order_num	order_date	customer_num	
1001	05/20/1990	104	***orders*** *table (detail)*
1002	05/21/1990	101	
1003	05/22/1990	104	
1004	05/22/1990	106	

Figure A-2 ***Tables joined by the*** **customer_num** ***column***

The **customer** table contains a **customer_num** column that holds a number identifying a customer, along with columns for the customer's name, company, address, and telephone number. For example, the row with information about Anthony Higgins contains the number 104 in the **customer_num** column. The **orders** table also contains a **customer_num** column that stores the number of the customer who placed a particular order.

According to Figure A-2, customer 104 (Anthony Higgins) has placed two orders since his customer number appears in two rows of the **orders** table. Since the join relationship lets you select information from both tables, you can retrieve Anthony Higgins' name and address and information about his orders at the same time.

Join Columns in the orders and items Tables

The **orders** and **items** tables are linked by an **order_num** column that contains an identification number for each order. If an order includes several items, the same order number appears in several rows of the **items** table. Figure A-3 shows this relationship.

***orders** table (detail)*

order_num	order_date	customer_num
1001	05/20/1990	104
1002	05/21/1990	101
1003	05/22/1990	104

***items** table (detail)*

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ

Figure A-3 ***Tables joined by the* order_num *column***

Join Columns in the items and stock Tables

The **items** table and the **stock** table are joined by two columns: the **stock_num** column stores a stock number for an item, and the **manu_code** column stores a code that identifies the manufacturer. You need both the stock number and the manufacturer code to uniquely identify an item.

For example, the item with the stock number `1` and the manufacturer code `HRO` is a Hero baseball glove, while the item with the stock number `1` and the manufacturer code `HSK` is a Husky baseball glove.

The same stock number and manufacturer code can appear in more than one row of the **items** table, if the same item belongs to separate orders, as illustrated in Figure A-4.

***items** table (detail)*

item_num	order_num	stock_num	manu_code
1	1001	1	HRO
1	1002	4	HSK
2	1002	3	HSK
1	1003	9	ANZ
2	1003	8	ANZ
3	1003	5	ANZ
1	1004	1	HRO

***stock** table (detail)*

stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
SMTbaseball gloves		

Figure A-4 ***Tables joined by the* stock_num *and* manu_code *columns***

Join Columns in the stock and catalog Tables

The **catalog** table and the **stock** table are joined by two columns: the **stock_num** column stores a stock number for an item, and the **manu_code** column stores a code that identifies the manufacturer. You need both of these columns to uniquely identify an item. Figure A-5 shows this relationship.

stock table (detail)

stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

catalog table (detail)

catalog_num	stock_num	manu_code
10001	1	HRO
10002	1	HSK
10003	1	SMT
10004	2	HRO

***Figure A-5** Tables joined by the* stock_num *and* manu_code *columns*

Join Columns in the stock and manufact Tables

The **stock** table and the **manufact** table are joined by the **manu_code** column. The same manufacturer code can appear in more than one row of the **stock** table if the manufacturer produces more than one piece of equipment. This relationship is illustrated in Figure A-6.

stock table (detail)

stock_num	manu_code	description
1	HRO	baseball gloves
1	HSK	baseball gloves
1	SMT	baseball gloves

manufact table (detail)

manu_code	manu_name
NRG	Norge
HSK	Husky
HRO	Hero

***Figure A-6** Tables joined by the* manu_code *column*

Join Columns in the cust_calls and customer Tables

The **cust_calls** table and the **customer** table are joined by the **customer_num** column. The same customer number can appear in more than one row of the **cust_calls** table if the customer calls the distributor more than once with a problem or question. This relationship is illustrated in Figure A-7.

***customer** table (detail)*

customer_num	fname	lname
101	Ludwig	Pauli
102	Carole	Sadler
103	Philip	Currie
104	Anthony	Higgins
105	Raymond	Vector
106	George	Watson

***cust_calls** table (detail)*

customer_num	call_dtime	user_id
106	1990-06-12 08:20	maryj
127	1990-07-31 14:30	maryj
116	1990-11-28 13:34	mannyh
116	1989-12-21 11:24	mannyh

Figure A-7 ***Tables joined by the*** **customer_num** ***column***

Join Columns in the state and customer Tables

The **state** table and the **customer** table are joined by a column that contains the state code. This column is called **code** in the **state** table and **state** in the **customer** table. If several customers live in the same state, the same state code will appear in several rows of the table, as shown in Figure A-8.

customer_num	fname	lname	...	state
101	Ludwig	Pauli	...	CA
102	Carole	Sadler	...	CA
103	Philip	Currie	...	CA

***customer** table (detail)*

code	sname
AK	Alaska
AL	Alabama
AR	Arkansas
AZ	Arizona
CA	California

***state** table (detail)*

***Figure A-8** Tables joined by the* **state/code** *column*

Data in the Demonstration Database

The tables that follow display the data in the stores demonstration database.

customer Table

customer_num	fname	lname	company	address1	address2	city	state	zipcode	phone
101	Ludwig	Pauli	All Sports Supplies	213 Erstwild Court		Sunnyvale	CA	94086	408-789-8075
102	Carole	Sadler	Sports Spot	785 Geary St		San Francisco	CA	94117	415-822-1289
103	Philip	Currie	Phil's Sports	654 Poplar	P. O. Box 3498	Palo Alto	CA	94303	415-328-4543
104	Anthony	Higgins	Play Ball!	East Shopping Cntr.	422 Bay Road	Redwood City	CA	94026	415-368-1100
105	Raymond	Vector	Los Altos Sports	1899 La Loma Drive		Los Altos	CA	94022	415-776-3249
106	George	Watson	Watson & Son	1143 Carver Place		Mountain View	CA	94063	415-389-8789
107	Charles	Ream	Athletic Supplies	41 Jordan Avenue		Palo Alto	CA	94304	415-356-9876
108	Donald	Quinn	Quinn's Sports	587 Alvarado		Redwood City	CA	94063	415-544-8729
109	Jane	Miller	Sport Stuff	Mayfair Mart	7345 Ross Blvd.	Sunnyvale	CA	94086	408-723-8789
110	Roy	Jaeger	AA Athletics	520 Topaz Way		Redwood City	CA	94062	415-743-3611
111	Frances	Keyes	Sports Center	3199 Sterling Court		Sunnyvale	CA	94085	408-277-7245
112	Margaret	Lawson	Runners & Others	234 Wyandotte Way		Los Altos	CA	94022	415-887-7235
113	Lana	Beatty	Sportstown	654 Oak Grove		Menlo Park	CA	94025	415-356-9982
114	Frank	Albertson	Sporting Place	947 Waverly Place		Redwood City	CA	94062	415-886-6677
115	Alfred	Grant	Gold Medal Sports	776 Gary Avenue		Menlo Park	CA	94025	415-356-1123
116	Jean	Parmelee	Olympic City	1104 Spinosa Drive		Mountain View	CA	94040	415-534-8822
117	Arnold	Sipes	Kids Korner	850 Lytton Court		Redwood City	CA	94063	415-245-4578
118	Dick	Baxter	Blue Ribbon Sports	5427 College		Oakland	CA	94609	415-655-0011
119	Bob	Shorter	The Triathletes Club	2405 Kings Highway		Cherry Hill	NJ	08002	609-663-6079
120	Fred	Jewell	Century Pro Shop	6627 N. 17th Way		Phoenix	AZ	85016	602-265-8754
121	Jason	Wallack	City Sports	Lake Biltmore Mall	350 W. 23rd Street	Wilmington	DE	19898	302-366-7511
122	Cathy	O'Brian	The Sporting Life	543 Nassau Street		Princeton	NJ	08540	609-342-0054
123	Marvin	Hanlon	Bay Sports	10100 Bay Meadows Rd	Suite 1020	Jacksonviille	FL	32256	904-823-4239
124	Chris	Putnum	Putnum's Putters	4715 S.E. Adams Blvd	Suite 909C	Bartlesville	OK	74006	918-355-2074
125	James	Henry	Total Fitness Sports	1450 Commonwealth Av		Brighton	MA	02135	617-232-4159
126	Eileen	Neelie	Neelie's Discount Sp	2539 South Utica Str		Denver	CO	80219	303-936-7731
127	Kim	Satifer	Big Blue Bike Shop	Blue Island Square	12222 Gregory Street	Blue Island	NY	60406	312-944-5691
128	Frank	Lessor	Phoenix University	Athletic Department	1817 N. Thomas Road	Phoenix	AZ	85008	602-533-1817

items Table (1 of 2)

item_num	order_num	stock_num	manu_code	quantity	total_price
1	1001	1	HRO	1	250.00
1	1002	4	HSK	1	960.00
2	1002	3	HSK	1	240.00
1	1003	9	ANZ	1	20.00
2	1003	8	ANZ	1	840.00
3	1003	5	ANZ	5	99.00
1	1004	1	HRO	1	250.00
2	1004	2	HRO	1	126.00
3	1004	3	HSK	1	240.00
4	1004	1	HSK	1	800.00
1	1005	5	NRG	10	280.00
2	1005	5	ANZ	10	198.00
3	1005	6	SMT	1	36.00
4	1005	6	ANZ	1	48.00
1	1006	5	SMT	5	125.00
2	1006	5	NRG	5	140.00
3	1006	5	ANZ	5	99.00
4	1006	6	SMT	1	36.00
5	1006	6	ANZ	1	48.00
1	1007	1	HRO	1	250.00
2	1007	2	HRO	1	126.00
3	1007	3	HSK	1	240.00
4	1007	4	HRO	1	480.00
5	1007	7	HRO	1	600.00
1	1008	8	ANZ	1	840.00
2	1008	9	ANZ	5	100.00
1	1009	1	SMT	1	450.00
1	1010	6	SMT	1	36.00
2	1010	6	ANZ	1	48.00
1	1011	5	ANZ	5	99.00
1	1012	8	ANZ	1	840.00
2	1012	9	ANZ	10	200.00
1	1013	5	ANZ	1	19.80
2	1013	6	SMT	1	36.00
3	1013	6	ANZ	1	48.00
4	1013	9	ANZ	2	40.00
1	1014	4	HSK	1	960.00
2	1014	4	HRO	1	480.00
1	1015	1	SMT	1	450.00
1	1016	101	SHM	2	136.00
2	1016	109	PRC	3	90.00
3	1016	110	HSK	1	308.00
4	1016	114	PRC	1	120.00
1	1017	201	NKL	4	150.00
2	1017	202	KAR	1	230.00
3	1017	301	SHM	2	204.00
1	1018	307	PRC	2	500.00

items Table (2 of 2)

item_num	order_num	stock_num	manu_code	quantity	total_price
2	1018	302	KAR	3	15.00
3	1018	110	PRC	1	236.00
4	1018	5	SMT	4	100.00
5	1018	304	HRO	1	280.00
1	1019	111	SHM	3	1499.97
1	1020	204	KAR	2	90.00
2	1020	301	KAR	4	348.00
1	1021	201	NKL	2	75.00
2	1021	201	ANZ	3	225.00
3	1021	202	KAR	3	690.00
4	1021	205	ANZ	2	624.00
1	1022	309	HRO	1	40.00
2	1022	303	PRC	2	96.00
3	1022	6	ANZ	2	96.00
1	1023	103	PRC	2	40.00
2	1023	104	PRC	2	116.00
3	1023	105	SHM	1	80.00
4	1023	110	SHM	1	228.00
5	1023	304	ANZ	1	170.00
6	1023	306	SHM	1	190.00

orders Table

order_num	order_date	customer_num	ship_instruct	backlog	po_num	ship_date	ship_weight	ship_charge	paid_date
1001	05/20/1990	104	express	n	B77836	06/01/1990	20.40	10.00	07/22/1990
1002	05/21/1990	101	PO on box; deliver back door only	n	9270	05/26/1990	50.60	15.30	06/03/1990
1003	05/22/1990	104	express	n	B77890	05/23/1990	35.60	10.80	06/14/1990
1004	05/22/1990	106	ring bell twice	y	8006	05/30/1990	95.80	19.20	
1005	05/24/1990	116	call before delivery	n	2865	06/09/1990	80.80	16.20	06/21/1990
1006	05/30/1990	112	after 10 am	y	Q13557		70.80	14.20	
1007	05/31/1990	117		n	278693	06/05/1990	125.90	25.20	
1008	06/07/1990	110	closed Monday	y	LZ230	07/06/1990	45.60	13.80	07/21/1990
1009	06/14/1990	111	next door to grocery	n	4745	06/21/1990	20.40	10.00	08/21/1990
1010	06/17/1990	115	deliver 776 King St. if no answer	n	429Q	06/29/1990	40.60	12.30	08/22/1990
1011	06/18/1990	104	express	n	B77897	07/03/1990	10.40	5.00	08/29/1990
1012	06/18/1990	117		n	278701	06/29/1990	70.80	14.20	
1013	06/22/1990	104	express	n	B77930	07/10/1990	60.80	12.20	07/31/1990
1014	06/25/1990	106	ring bell, kick door loudly	n	8052	07/03/1990	40.60	12.30	07/10/1990
1015	06/27/1990	110	closed Mondays	n	MA003	07/16/1990	20.60	6.30	08/31/1990
1016	06/29/1990	119	delivery entrance off Camp St.	n	PC6782	07/12/1990	35.00	11.80	
1017	07/09/1990	120	North side of clubhouse	n	DM354331	07/13/1990	60.00	18.00	
1018	07/10/1990	121	SW corner of Biltmore Mall	n	S22942	07/13/1990	70.50	20.00	08/06/1990
1019	07/11/1990	122	closed til noon Mondays	n	Z55709	07/16/1990	90.00	23.00	08/06/1990
1020	07/11/1990	123	express	n	W2286	07/16/1990	14.00	8.50	09/20/1990
1021	07/23/1990	124	ask for Elaine	n	C3288	07/25/1990	40.00	12.00	08/22/1990
1022	07/24/1990	126	express	n	W9925	07/30/1990	15.00	13.00	09/02/1990
1023	07/24/1990	127	no deliveries after 3 p.m.	n	KF2961	07/30/1990	60.00	18.00	08/22/1990

stock Table (1 of 2)

stock_num	manu_code	description	unit_price	unit	unit_descr
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
5	SMT	tennis racquet	25.00	each	each
5	ANZ	tennis racquet	19.80	each	each
6	SMT	tennis ball	36.00	case	24 cans/case
6	ANZ	tennis ball	48.00	case	24 cans/case
7	HRO	basketball	600.00	case	24/case
8	ANZ	volleyball	840.00	case	24/case
9	ANZ	volleyball net	20.00	each	each
101	PRC	bicycle tires	88.00	box	4/box
101	SHM	bicycle tires	68.00	box	4/box
102	SHM	bicycle brakes	220.00	case	4 sets/case
102	PRC	bicycle brakes	480.00	case	4 sets/case
103	PRC	front derailleur	20.00	each	each
104	PRC	rear derailleur	58.00	each	each
105	PRC	bicycle wheels	53.00	pair	pair
105	SHM	bicycle wheels	80.00	pair	pair
106	PRC	bicycle stem	23.00	each	each
107	PRC	bicycle saddle	70.00	pair	pair
108	SHM	crankset	45.00	each	each
109	PRC	pedal binding	30.00	case	6 pairs/case
109	SHM	pedal binding	200.00	case	4 pairs/case
110	PRC	helmet	236.00	case	4/case
110	ANZ	helmet	244.00	case	4/case
110	SHM	helmet	228.00	case	4/case
110	HRO	helmet	260.00	case	4/case
110	HSK	helmet	308.00	case	4/case
111	SHM	10-spd, assmbld	499.99	each	each
112	SHM	12-spd, assmbld	549.00	each	each
113	SHM	18-spd, assmbld	685.90	each	each
114	PRC	bicycle gloves	120.00	case	10 pairs/case

stock Table (2 of 2)

stock_num	manu_code	description	unit_price	unit	unit_descr
201	NKL	golf shoes	37.50	each	each
201	ANZ	golf shoes	75.00	each	each
201	KAR	golf shoes	90.00	each	each
202	NKL	metal woods	174.00	case	2 sets/case
202	KAR	std woods	230.00	case	2 sets/case
203	NKL	irons/wedges	670.00	case	2 sets/case
204	KAR	putter	45.00	each	each
205	NKL	3 golf balls	312.00	case	24/case
205	ANZ	3 golf balls	312.00	case	24/case
205	HRO	3 golf balls	312.00	case	24/case
301	NKL	running shoes	97.00	each	each
301	HRO	running shoes	42.50	each	each
301	SHM	running shoes	102.00	each	each
301	PRC	running shoes	75.00	each	each
301	KAR	running shoes	87.00	each	each
301	ANZ	running shoes	95.00	each	each
302	HRO	ice pack	4.50	each	each
302	KAR	ice pack	5.00	each	each
303	PRC	socks	48.00	box	24 pairs/box
303	KAR	socks	36.00	box	24 pair/box
304	ANZ	watch	170.00	box	10/box
304	HRO	watch	280.00	box	10/box
305	HRO	first-aid kit	48.00	case	4/case
306	PRC	tandem adapter	160.00	each	each
306	SHM	tandem adapter	190.00	each	each
307	PRC	infant jogger	250.00	each	each
308	PRC	twin jogger	280.00	each	each
309	HRO	ear drops	40.00	case	20/case
309	SHM	ear drops	40.00	case	20/case
310	SHM	kick board	80.00	case	10/case
310	ANZ	kick board	89.00	case	12/case
311	SHM	water gloves	48.00	box	4 pairs/box
312	SHM	racer goggles	96.00	box	12/box
312	HRO	racer goggles	72.00	box	12/box
313	SHM	swim cap	72.00	box	12/box
313	ANZ	swim cap	60.00	box	12/box

catalog Table (1 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10001	1	HRO8	Brown leather. Specify first baseman's or infield/outfield style. Specify right- or left-handed.	<BYTE value>	Your First Season's Baseball Glove
10002	1	HSK	Babe Ruth signature glove. Black leather. Infield/outfield style. Specify right- or left-handed	<BYTE value>	All-Leather, Hand-Stitched, Deep-Pockets, Sturdy Webbing that Won't Let Go
10003	1	SMT	Catcher's mitt. Brown leather. Specify right- or left-handed.	<BYTE value>	A Sturdy Catcher's Mitt With the Perfec Pocket
10004	2	HRO	Jackie Robinson signature glove. Highest Professional quality, used by National League.	<BYTE value>	Highest Quality Ball Available, from the Hand-Stitching to the Robinson Signature
10005	3	HSK	Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.	<BYTE value>	High-Technology Design Expands the Sweet Spot
10006	3	SHM	Aluminum. Blue with black tape. 31", 20 oz or 22 oz; 32", 21 oz or 23 oz; 33", 22 oz or 24 oz;	<BYTE value>	Durable Aluminum for High School an Collegiate Athletes
10007	4	HSK	Norm Van Brocklin signature style.	<BYTE value>	Quality Pigskin with Norm Van Brockli Signature
10008	4	HRO	NFL-Style pigskin.	<BYTE value>	Highest Quality Football for High School and Collegiate Competitions
10009	5	NRG	Graphite frame. Synthetic strings.	<BYTE value>	Wide Body Amplifies Your Natural Abilities by Providing More Power Through Aerodynamic Design
10010	5	SMT	Aluminum frame. Synthetic strings	<BYTE value>	Mid-Sized Racquet For the Improving Player
10011	5	ANZ	Wood frame, cat-gut strings.	<BYTE value>	Antique Replica of Classic Wooden Racquet Built with Cat-Gut Strings
10012	6	SMT	Soft yellow color for easy visibility in sunlight or artificial light	<BYTE value>	High-Visibility Tennis, Day or Night
10013	6	ANZ	Pro-core. Available in neon yellow, green, and pink.	<BYTE value>	Durable Construction Coupled with th Brightest Colors Available
10014	7	HRO	Indoor. Classic NBA style. Brown leather.	<BYTE value>	Long-Life Basketballs for Indoor Gymnasiums
10015	8	ANZ	Indoor. Finest leather. Professional quality.	<BYTE value>	Professional Volleyballs for Indoor Competitions
10016	9	ANZ	Steel eyelets. Nylon cording. Double-stitched. Sanctioned by the National Athletic Congress	<BYTE value>	Sanctioned Volleyball Netting for Indoor Professional and Collegiate Competition

catalog Table (2 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10017	101	PRC	Reinforced, hand-finished tubular. Polyurethane belted. Effective against punctures. Mixed tread for super wear and road grip.	<BYTE value>	Ultimate in Puncture Protection, Tires Designed for In-City Riding
10018	101	SHM	Durable nylon casing with butyl tube for superior air retention. Center-ribbed tread with herringbone side. Coated sidewalls resist abrasion.	<BYTE value>	The Perfect Tire for Club Rides or Training
10019	102	SHM	Thrust bearing and coated pivot washer/spring sleeve for smooth action. Slotted levers with soft gum hoods. Two-tone paint treatment. Set includes calipers, levers, and cables.	<BYTE value>	Thrust-Bearing and Spring-Sleeve Brak Set Guarantees Smooth Action
10020	102	PRC	Computer-aided design with low-profile pads. Cold-forged alloy calipers and beefy caliper bushing. Aero levers. Set includes calipers, levers, and cables	<BYTE value>	Computer Design Delivers Rigid Yet Vibration-Free Brakes
10021	103	PRC	Compact leading-action design enhances shifting. Deep cage for super-small granny gears. Extra strong construction to resist off-road abuse.	<BYTE value>	Climb Any Mountain: ProCycle's Fron Derailleur Adds Finesse to Your ATB
10022	104	PRC	Floating trapezoid geometry with extra thick parallelogram arms. 100-tooth capacity. Optimum alignment with any freewheel.	<BYTE value>	Computer-Aided Design Engineers 100-Tooth Capacity Into ProCycle's Rea Derailleur
10023	105	PRC	Front wheels laced with 15g spokes in a 3-cross pattern. Rear wheels laced with 14g spikes in a 3-cross pattern.	<BYTE value>	Durable Training Wheels That Hold True Under Toughest Conditions
10024	105	SHM	Polished alloy. Sealed-bearing, quick-release hubs. Double-butted. Front wheels are laced 15g/2-cross. Rear wheels are laced 15g/3-cross.	<BYTE value>	Extra Lightweight Wheels for Training or High-Performance Touring
10025	106	PRC	Hard anodized alloy with pearl finish. 6mm hex bolt hardware. Available in lengths of 90-140mm in 10mm increments.	<BYTE value>	ProCycle Stem with Pearl Finish
10026	107	PRC	Available in three styles: Mens racing; Mens touring; and Womens. Anatomical gel construction with lycra cover. Black or black/hot pink.	<BYTE value>	The Ultimate In Riding Comfort, Light weight With Anatomical Support

catalog Table (3 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10027	108	SHM	Double or triple crankset with choice of chainrings. For double crankset, chainrings from 38-54 teeth. For triple crankset, chainrings from 24-48 teeth.	<BYTE value>	Customize Your Mountain Bike With Extra-Durable Crankset
10028	109	PRC	Steel toe clips with nylon strap. Extra wide at buckle to reduce pressure.	<BYTE value>	Classic Toeclip Improved To Prevent Soreness At Clip Buckle
10029	109	SHM	Ingenious new design combines button on sole of shoe with slot on a pedal plate to give riders new options in riding efficiency. Choose full or partial locking. Four plates mean both top and bottom of pedals are slotted—no fishing around when you want to engage full power. Fast unlocking ensures safety when maneuverability is paramount.	<BYTE value>	Ingenious Pedal/Clip Design Delivers Maximum Power And Fast Unlocking
10030	110	PRC	Super-lightweight. Meets both ANZI and Snell standards for impact protection. 7.5 oz. Quick-release shadow buckle.	<BYTE value>	Feather-Light, Quick-Release, Maximum Protection Helmet
10031	110	ANZ	No buckle so no plastic touches your chin. Meets both ANZI and Snell standards for impact protection. 7.5 oz. Lycra cover.	<BYTE value>	Minimum Chin Contact, Feather-Light, Maximum Protection Helmet
10032	110	SHM	Dense outer layer combines with softer inner layer to eliminate the mesh cover, no snagging on brush. Meets both ANZI and Snell standards for impact protection. 8.0 oz.	<BYTE value>	Mountain Bike Helmet: Smooth Cover Eliminates the Worry of Brush Snags But Delivers Maximum Protection
10033	110	HRO	Newest ultralight helmet uses plastic shell. Largest ventilation channels of any helmet on the market. 8.5 oz.	<BYTE value>	Lightweight Plastic with Vents Assures Cool Comfort Without Sacrificing Protection
10034	110	HSK	Aerodynamic (teardrop) helmet covered with anti-drag fabric. Credited with shaving 2 seconds/mile from winner's time in Tour de France time-trial. 7.5 oz.	<BYTE value>	Teardrop Design Used by Yellow Jerseys, You Can Time the Difference
10035	111	SHM	Light-action shifting 10 speed. Designed for the city commuter with shock-absorbing front fork and drilled eyelets for carry-all racks or bicycle trailers. Internal wiring for generator lights. 33 lbs.	<BYTE value>	Fully Equipped Bicycle Designed for the Serious Commuter Who Mixes Business With Pleasure

catalog Table (4 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10036	112	SHM	Created for the beginner enthusiast. Ideal for club rides and light touring. Sophisticated triple-butted frame construction. Precise index shifting. 28 lbs.	<BYTE value>	We Selected the Ideal Combination of Touring Bike Equipment, Then Turned It Into This Package Deal: High-Performance on the Roads, Maximum Pleasure Everywhere
10037	113	SHM	Ultra-lightweight. Racing frame geometry built for aerodynamic handlebars. Cantilever brakes. Index shifting. High-performance gearing. Quick-release hubs. Disk wheels. Bladed spokes.	<BYTE value>	Designed for the Serious Competitor, The Complete Racing Machine
10038	114	PRC	Padded leather palm and stretch mesh merged with terry back; Available in tan, black, and cream. Sizes S, M, L, XL.	<BYTE value>	Riding Gloves For Comfort and Protection
10039	201	NKL	Designed for comfort and stability. Available in white & blue or white & brown. Specify size.	<BYTE value>	Full-Comfort, Long-Wearing Golf Shoes for Men and Women
10040	201	ANZ	Guaranteed waterproof. Full leather upper. Available in white, bone, brown, green, and blue. Specify size.	<BYTE value>	Waterproof Protection Ensures Maximum Comfort and Durability In All Climates
10041	201	KAR	Leather and leather mesh for maximum ventilation. Waterproof lining to keep feet dry. Available in white & gray or white & ivory. Specify size.	<BYTE value>	Karsten's Top Quality Shoe Combines Leather and Leather Mesh
10042	202	NKL	Complete starter set utilizes gold shafts. Balanced for power.	<BYTE value>	Starter Set of Woods, Ideal for High School and Collegiate Classes
10043	202	KAR	Full set of woods designed for precision control and power performance.	<BYTE value>	High-Quality Woods Appropriate for High School Competitions or Serious Amateurs
10044	203	NKL	Set of eight irons includes 3 through 9 irons and pitching wedge. Originally priced at $489.00.	<BYTE value>	Set of Irons Available From Factory at Tremendous Savings: Discontinued Line.
10045	204	KAR	Ideally balanced for optimum control. Nylon-covered shaft.	<BYTE value>	High-Quality Beginning Set of Irons Appropriate for High School Competitions
10046	205	NKL	Fluorescent yellow.	<BYTE value>	Long Drive Golf Balls: Fluorescent Yellow
10047	205	ANZ	White only.	<BYTE value>	Long Drive Golf Balls: White
10048	205	HRO	Combination fluorescent yellow and standard white.	<BYTE value>	HiFlier Golf Balls: Case Includes Fluorescent Yellow and Standard White

catalog Table (5 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10049	301	NKL	Super shock-absorbing gel pads disperse vertical energy into a horizontal plane for extraordinary cushioned comfort. Great motion control. Mens only. Specify size.	<BYTE value>	Maximum Protection For High-Mileage Runners
10050	301	HRO	Engineered for serious training with exceptional stability. Fabulous shock absorption. Great durability. Specify mens/womens, size.	<BYTE value>	Pronators and Supinators Take Heart: A Serious Training Shoe For Runners Who Need Motion Control
10051	301	SHM	For runners who log heavy miles and need a durable, supportive, stable platform. Mesh/synthetic upper gives excellent moisture dissipation. Stability system uses rear antipronation platform and forefoot control plate for extended protection during high-intensity training. Specify mens/womens, size.	<BYTE value>	The Training Shoe Engineered for Marathoners and Ultra-Distance Runners
10052	301	PRC	Supportive, stable racing flat. Plenty of forefoot cushioning with added motion control. Womens only. D widths available. Specify size.	<BYTE value>	A Woman's Racing Flat That Combines Extra Forefoot Protection With a Slender Heel
10053	301	KAR	Anatomical last holds your foot firmly in place. Feather-weight cushioning delivers the responsiveness of a racing flat. Specify mens/womens, size.	<BYTE value>	Durable Training Flat That Can Carry You Through Marathon Miles
10054	301	ANZ	Cantilever sole provides shock absorption and energy rebound. Positive traction shoe with ample toe box. Ideal for runners who need a wide shoe. Available in mens and womens. Specify size.	<BYTE value>	Motion Control, Protection, and Extra Toebox Room
10055	302	KAR	Re-usable ice pack with velcro strap. For general use. Velcro strap allows easy application to arms or legs.	<BYTE value>	Finally, An Ice Pack for Achilles Injuries and Shin Splints that You Can Take to the Office
10056	303	PRC	Neon nylon. Perfect for running or aerobics. Indicate color: Fluorescent pink, yellow, green, and orange.	<BYTE value>	Knock Their Socks Off With YOUR Socks!
10057	303	KAR	100% nylon blend for optimal wicking and comfort. We've taken out the cotton to eliminate the risk of blisters and reduce the opportunity for infection. Specify mens or womens.	<BYTE value>	100% Nylon Blend Socks - No Cotton!

catalog Table (6 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10058	304	ANZ	Provides time, date, dual display of lap/cumulative splits, 4-lap memory, 10hr count-down timer, event timer, alarm, hour chime, waterproof to 50m, velcro band.	<BYTE value>	Athletic Watch w/4-Lap Memory
10059	304	HRO	Split timer, waterproof to 50m. Indicate color: Hot pink, mint green, space black.	<BYTE value>	Waterproof Triathlete Watch In Competition Colors
10060	305	HRO	Contains ace bandage, anti-bacterial cream, alcohol cleansing pads, adhesive bandages of assorted sizes, and instant-cold pack.	<BYTE value>	Comprehensive First-Aid Kit Essential for Team Practices, Team Traveling
10061	306	PRC	Converts a standard tandem bike into an adult/child bike. User-tested Assembly Instructions	<BYTE value>	Enjoy Bicycling With Your Child On a Tandem; Make Your Family Outing Safer
10062	306	SHM	Converts a standard tandem bike into an adult/child bike. Lightweight model.	<BYTE value>	Consider a Touring Vacation For the Entire Family: A Lightweight, Touring Tandem for Parent and Child
10063	307	PRC	Allows mom or dad to take the baby out, too. Fits children up to 21 pounds. Navy blue with black trim.	<BYTE value>	Infant Jogger Keeps A Running Family Together
10064	308	PRC	Allows mom or dad to take both children! Rated for children up to 18 pounds.	<BYTE value>	As Your Family Grows, Infant Jogger Grows With You
10065	309	HRO	Prevents swimmer's ear.	<BYTE value>	Swimmers Can Prevent Ear Infection All Season Long
10066	309	SHM	Extra-gentle formula. Can be used every day for prevention or treatment of swimmer's ear.	<BYTE value>	Swimmer's Ear Drops Specially Formulated for Children
10067	310	SHM	Blue heavy-duty foam board with Shimara or team logo.	<BYTE value>	Exceptionally Durable, Compact Kickboard for Team Practice
10068	310	ANZ	White. Standard size.	<BYTE value>	High-Quality Kickboard
10069	311	SHM	Swim gloves. Webbing between fingers promotes strengthening of arms. Cannot be used in competition.	<BYTE value>	Hot Training Tool - Webbed Swim Gloves Build Arm Strength and Endurance
10070	312	SHM	Hydrodynamic egg-shaped lens. Ground-in anti-fog elements; Available in blue or smoke.	<BYTE value>	Anti-Fog Swimmer's Goggles: Quantity Discount.
10071	312	HRO	Durable competition-style goggles. Available in blue, grey, or white.	<BYTE value>	Swim Goggles: Traditional Rounded Lens For Greater Comfort.

catalog Table (7 of 7)

catalog_num	stock_num	manu_code	cat_descr	cat_picture	cat_advert
10072	313	SHM	Silicone swim cap. One size. Available in white, silver, or navy. Team Logo Imprinting Available	<BYTE value>	Team Logo Silicone Swim Cap
10073	313	ANZ	Silicone swim cap. Squared-off top. One size. White.	<BYTE value>	Durable Squared-off Silicone Swim Cap
10074	302	HRO	Re-usable ice pack. Store in the freezer for instant first-aid. Extra capacity to accommodate water and ice.	<BYTE value>	Water Compartment Combines With Ice to Provide Optimal Orthopedic Treatment

cust_calls Table

customer_num	call_dtime	user_id	call_code	call_descr	res_dtime	res_descr
106	1990-06-12 8:20	maryj	D	Order was received, but two of the cans of ANZ tennis balls within the case were empty	1990-06-12 8:25	Authorized credit for two cans to customer, issued apology. Called ANZ buyer to report the QA problem.
110	1990-07-07 10:24	richc	L	Order placed one month ago (6/7) not received.	1990-07-07 10:30	Checked with shipping (Ed Smith). Order sent yesterday- we were waiting for goods from ANZ. Next time will call with delay if necessary.
119	1990-07-01 15:00	richc	B	Bill does not reflect credit from previous order	1990-07-02 8:21	Spoke with Jane Akant in Finance. She found the error and is sending new bill to customer
121	1990-07-10 14:05	maryj	O	Customer likes our merchandise. Requests that we stock more types of infant joggers. Will call back to place order.	1990-07-10 14:06	Sent note to marketing group of interest in infant joggers
127	1990-07-31 14:30	maryj	I	Received Hero watches (item # 304) instead of ANZ watches		Sent memo to shipping to send ANZ item 304 to customer and pickup HRO watches. Should be done tomorrow, 8/1
116	1989-11-28 13:34	mannyn	I	Received plain white swim caps (313 ANZ) instead of navy with team logo (313 SHM)	1989-11-28 16:47	Shipping found correct case in warehouse and express mailed it in time for swim meet.
116	1989-12-21 11:24	mannyn	I	Second complaint from this customer! Received two cases right-handed outfielder gloves (1 HRO) instead of one case lefties.	1989-12-27 08:19	Memo to shipping (Ava Brown) to send case of left-handed gloves, pick up wrong case; memo to billing requesting 5% discount to placate customer due to second offense and lateness of resolution because of holiday

manufact Table

manu_code	manu_name	lead_time
ANZ	Anza	5
HSK	Husky	5
HRO	Hero	4
NRG	Norge	7
SMT	Smith	3
SHM	Shimara	30
KAR	Karsten	21
NKL	Nikolus	8
PRC	ProCycle	9

state Table

code	sname	code	sname
AK	Alaska	MT	Montana
AL	Alabama	NE	Nebraska
AR	Arkansas	NC	North Carolina
AZ	Arizona	ND	North Dakota
CA	California	NH	New Hampshire
CT	Connecticut	NJ	New Jersey
CO	Colorado	NM	New Mexico
D.C.	DC	NV	Nevada
DE	Delaware	NY	New York
FL	Florida	OH	Ohio
GA	Georgia	OK	Oklahoma
HI	Hawaii	OR	Oregon
IA	Iowa	PA	Pennsylvania
ID	Idaho	PR	Puerto Rico
IL	Illinois	RI	Rhode Island
IN	Indiana	SC	South Carolina
KS	Kansas	SD	South Dakota
KY	Kentucky	TN	Tennessee
LA	Louisiana	TX	Texas
MA	Massachusetts	UT	Utah
MD	Maryland	VA	Virginia
ME	Maine	VT	Vermont
MI	Michigan	WA	Washington
MN	Minnesota	WI	Wisconsin
MO	Missouri	WV	West Virginia
MS	Mississippi	WY	Wyoming

The Demonstration Application

The following pages contain the form specifications, **INFORMIX-4GL** source code modules, and help message source file for the **demo4.4ge** demonstration application. The application is not complete, and some of the functions called by the menus are merely "dead ends."

File Name	Description
custform.per	Form for displaying customer information
orderform.per	Form for entering an order
state_list.per	Form for displaying a list of states
stock_sel.per	Form for displaying a list of stock items
d4_globals.4gl	Module containing global definitions
d4_main.4gl	Module containing MAIN routine
d4_cust.4gl	Module handling the **Customer** option
d4_orders.4gl	Module handling the **Orders** option
d4_stock.4gl	Module handling the **Stock** option
d4_report.4gl	Module handling the **Report** option
d4_demo.4gl	Module handling hidden sample source code option
helpdemo.src	Source file for help messages

custform.per

```
DATABASE stores

SCREEN
{

                           Customer Form

        Number      :[f000       ]
        Owner Name  :[f001           ][f002           ]
        Company     :[f003                ]
        Address     :[f004                ]
                     [f005                ]
        City        :[f006           ] State:[a0] Zipcode:[f007 ]
        Telephone   :[f008              ]

}

TABLES
customer

ATTRIBUTES
f000 = customer.customer_num, NOENTRY;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, UPSHIFT;
f007 = customer.zipcode;
f008 = customer.phone, PICTURE = "###-###-#### XXXXX";
```

orderform.per

```
DATABASE stores

SCREEN
{
 ------------------------------------------------------------------------------
                             ORDER FORM
 ------------------------------------------------------------------------------
 Customer Number:[f000       ]  Contact Name:[f001           ][f002           ]
    Company Name:[f003                ]
         Address:[f004                ][f005                 ]
            City:[f006           ] State:[a0] Zip Code:[f007 ]
       Telephone:[f008              ]
 ------------------------------------------------------------------------------
 Order No:[f009       ]    Order Date:[f010      ]    PO Number:[f011      ]
        Shipping Instructions:[f012                                    ]
 ------------------------------------------------------------------------------
 Item No.  Stock No.  Code    Description    Quantity    Price       Total
 [f013  ]  [f014  ]  [a1 ] [f015            ] [f016  ] [f017     ] [f018       ]
 [f013  ]  [f014  ]  [a1 ] [f015            ] [f016  ] [f017     ] [f018       ]
 [f013  ]  [f014  ]  [a1 ] [f015            ] [f016  ] [f017     ] [f018       ]
 [f013  ]  [f014  ]  [a1 ] [f015            ] [f016  ] [f017     ] [f018       ]
                 Running Total including Tax and Shipping Charges:[f019       ]
 ==============================================================================
}

TABLES
customer orders items stock

ATTRIBUTES
f000 = customer.customer_num;
f001 = customer.fname;
f002 = customer.lname;
f003 = customer.company;
f004 = customer.address1;
f005 = customer.address2;
f006 = customer.city;
a0 = customer.state, UPSHIFT;
f007 = customer.zipcode;
f008 = customer.phone, PICTURE = "###-###-#### XXXXX";

f009 = orders.order_num;
f010 = orders.order_date, DEFAULT = TODAY;
f011 = orders.po_num;
f012 = orders.ship_instruct;

f013 = items.item_num, NOENTRY;
f014 = items.stock_num;
a1 = items.manu_code, UPSHIFT;
f015 = stock.description, NOENTRY;
f016 = items.quantity;
f017 = stock.unit_price, NOENTRY;
f018 = items.total_price, NOENTRY;
f019 = formonly.t_price TYPE MONEY;

INSTRUCTIONS
SCREEN RECORD s_items[4](items.item_num, items.stock_num, items.manu_code,
        stock.description, items.quantity, stock.unit_price, items.total_price)
```

state_list.per

```
DATABASE stores

SCREEN
{
   State Selection

[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
[a0] [f000           ]
}

TABLES
state

ATTRIBUTES
a0 = state.code;
f000 = state.sname;

INSTRUCTIONS
DELIMITERS "  "
SCREEN RECORD s_state[7](state.*)
```

stock_sel.per

```
DATABASE stores

SCREEN
{
  [f018][f019][f020           ][f021           ][f022     ][f023           ]
  [f018][f019][f020           ][f021           ][f022     ][f023           ]
  [f018][f019][f020           ][f021           ][f022     ][f023           ]
}

TABLES
stock

ATTRIBUTES
f018 = FORMONLY.stock_num;
f019 = FORMONLY.manu_code;
f020 = FORMONLY.manu_name;
f021 = FORMONLY.description;
f022 = FORMONLY.unit_price;
f023 = FORMONLY.unit_descr;

INSTRUCTIONS
DELIMITERS "  "
SCREEN RECORD s_stock[3] (FORMONLY.stock_num THRU FORMONLY.unit_descr)
```

d4_globals.4gl

```
DATABASE stores

GLOBALS
   DEFINE
      p_customer RECORD LIKE customer.*,
      p_orders RECORD
            order_num LIKE orders.order_num,
            order_date LIKE orders.order_date,
            po_num LIKE orders.po_num,
            ship_instruct LIKE orders.ship_instruct
         END RECORD,
      p_items ARRAY[10] OF RECORD
            item_num LIKE items.item_num,
            stock_num LIKE items.stock_num,
            manu_code LIKE items.manu_code,
            description LIKE stock.description,
            quantity LIKE items.quantity,
            unit_price LIKE stock.unit_price,
            total_price LIKE items.total_price
         END RECORD,
      p_stock ARRAY[30] OF RECORD
            stock_num LIKE stock.stock_num,
            manu_code LIKE manufact.manu_code,
            manu_name LIKE manufact.manu_name,
            description LIKE stock.description,
            unit_price LIKE stock.unit_price,
            unit_descr LIKE stock.unit_descr
         END RECORD,
      p_state ARRAY[50] OF RECORD LIKE state.*,
      state_cnt, stock_cnt INTEGER,
      print_option CHAR(1)
END GLOBALS
```

d4_main.4gl

```
GLOBALS
   "d4_globals.4gl"


MAIN

   DEFER INTERRUPT
   OPTIONS
   HELP FILE "helpdemo"
   LET print_option = "s"
   CALL get_states()
   CALL get_stocks()

   CALL ring_menu()
   MENU "MAIN"
      COMMAND "Customer" "Enter and maintain customer data" HELP 101
         CALL customer()
         CALL ring_menu()
      COMMAND "Orders" "Enter and maintain orders" HELP 102
         CALL orders()
         CALL ring_menu()
      COMMAND "Stock" "Enter and maintain stock list" HELP 103
         CALL stock()
         CALL ring_menu()
      COMMAND "Reports" "Print reports and mailing labels" HELP 104
         CALL reports()
         CALL ring_menu()
      COMMAND key("!")
         CALL bang()
         CALL ring_menu()
         NEXT OPTION "Customer"
      COMMAND key("X")
         CALL demo()
         CALL ring_menu()
         NEXT OPTION "Customer"
      COMMAND "Exit" "Exit program and return to operating system" HELP 105
         CLEAR SCREEN
         EXIT PROGRAM
   END MENU
END MAIN


FUNCTION bang()
   DEFINE cmd CHAR(80),
      x CHAR(1)

   CALL clear_menu()
   LET x = "!"
   WHILE x = "!"
      PROMPT "!" FOR cmd
      RUN cmd
      PROMPT "Type RETURN to continue." FOR CHAR x
   END WHILE
END FUNCTION
```

```
FUNCTION mess(str, mrow)
   DEFINE str CHAR(80),
      mrow SMALLINT

   DISPLAY " ", str CLIPPED AT mrow,1
   SLEEP 3
   DISPLAY "" AT mrow,1
END FUNCTION


FUNCTION ring_menu()

   DISPLAY "---------------------------------------- ",
      "Type Control-W for MENU HELP -------" AT 4,2 ATTRIBUTE(MAGENTA)
END FUNCTION


FUNCTION clear_menu()

   DISPLAY "" AT 1,1
   DISPLAY "" AT 2,1
END FUNCTION


FUNCTION get_states()

   DECLARE c_state CURSOR FOR
      SELECT * FROM state
      ORDER BY sname
   LET state_cnt = 1
   FOREACH c_state INTO p_state[state_cnt].*
      LET state_cnt = state_cnt + 1
      IF state_cnt > 50 THEN
         EXIT FOREACH
      END IF
   END FOREACH
   LET state_cnt = state_cnt - 1
END FUNCTION


FUNCTION get_stocks()

   DECLARE stock_list CURSOR FOR
      SELECT stock_num, manufact.manu_code,
         manu_name, description, unit_price, unit_descr
      FROM stock, manufact
      WHERE stock.manu_code = manufact.manu_code
      ORDER BY stock_num
   LET stock_cnt = 1
   FOREACH stock_list INTO p_stock[stock_cnt].*
      LET stock_cnt = stock_cnt + 1
      IF stock_cnt > 30 THEN
         EXIT FOREACH
      END IF
   END FOREACH
   LET stock_cnt = stock_cnt - 1
END FUNCTION
```

d4_cust.4gl

```
GLOBALS
    "d4_globals.4gl"

FUNCTION customer()

   OPTIONS
      FORM LINE 7
   OPEN FORM customer FROM "custform"
   DISPLAY FORM customer
      ATTRIBUTE(MAGENTA)
   CALL ring_menu()
   CALL fgl_drawbox(3,30,3,43)
   CALL fgl_drawbox(3,61,8,7)
   CALL fgl_drawbox(11,61,8,7)
   LET p_customer.customer_num = NULL
   MENU "CUSTOMER"
      COMMAND "One-add" "Add a new customer to the database" HELP 201
         CALL add_customer(FALSE)
      COMMAND "Many-add" "Add several new customer to database" HELP 202
         CALL add_customer(TRUE)
      COMMAND "Find-cust" "Look up specific customer" HELP 203
         CALL query_customer(23)
         IF p_customer.customer_num IS NOT NULL THEN
            NEXT OPTION "Update-cust"
         END IF
      COMMAND "Update-cust" "Modify current customer information" HELP 204
         CALL update_customer()
         NEXT OPTION "Find-cust"
      COMMAND "Delete-cust" "Remove a customer from database" HELP 205
         CALL delete_customer()
         NEXT OPTION "Find-cust"
      COMMAND "Exit" "Return to MAIN Menu" HELP 206
         CLEAR SCREEN
         EXIT MENU
   END MENU
   OPTIONS
      FORM LINE 3
END FUNCTION


FUNCTION add_customer(repeat)
   DEFINE repeat INTEGER

   CALL clear_menu()
   MESSAGE "Press F1 or CTRL-F for field help; ",
      "F2 or CTRL-Z to return to menu"
   IF repeat THEN
      WHILE input_cust()
         ERROR "Customer data entered" ATTRIBUTE (GREEN)
      END WHILE
      CALL mess("Multiple insert completed -
 current screen values ignored", 23)
   ELSE
      IF input_cust() THEN
         ERROR "Customer data entered" ATTRIBUTE (GREEN)
      ELSE
         CLEAR FORM
         LET p_customer.customer_num = NULL
         ERROR "Customer addition aborted" ATTRIBUTE (RED, REVERSE)
      END IF
   END IF
```

```
END FUNCTION
FUNCTION input_cust()

   DISPLAY "Press ESC to enter new customer data" AT 1,1
   INPUT BY NAME p_customer.*
      AFTER FIELD state
         CALL statehelp()
         DISPLAY "Press ESC to enter new customer data", "" AT 1,1
      ON KEY (F1, CONTROL-F)
         CALL customer_help()
      ON KEY (F2, CONTROL-Z)
         LET int_flag = TRUE
         EXIT INPUT
   END INPUT
   IF int_flag THEN
      LET int_flag = FALSE
      RETURN(FALSE)
   END IF
   LET p_customer.customer_num = 0
   INSERT INTO customer VALUES (p_customer.*)
   LET p_customer.customer_num = SQLCA.SQLERRD[2]
   DISPLAY BY NAME p_customer.customer_num ATTRIBUTE(MAGENTA)
   RETURN(TRUE)
END FUNCTION


FUNCTION query_customer(mrow)
   DEFINE where_part CHAR(200),
      query_text CHAR(250),
      answer CHAR(1),
      mrow, chosen, exist SMALLINT

   CLEAR FORM
   CALL clear_menu()

   MESSAGE "Enter criteria for selection"
   CONSTRUCT where_part ON customer.* FROM customer.*
   MESSAGE ""
   IF int_flag THEN
      LET int_flag = FALSE
      CLEAR FORM
      ERROR "Customer query aborted" ATTRIBUTE(RED, REVERSE)
      LET p_customer.customer_num = NULL
      RETURN (p_customer.customer_num)
   END IF
   LET query_text = "select * from customer where ", where_part CLIPPED,
      " order by lname"
   PREPARE statement_1 FROM query_text
   DECLARE customer_set SCROLL CURSOR FOR statement_1
```

```
    OPEN customer_set
    FETCH FIRST customer_set INTO p_customer.*
    IF status = NOTFOUND THEN
       LET exist = FALSE
    ELSE
       LET exist = TRUE
       DISPLAY BY NAME p_customer.*
       MENU "BROWSE"
          COMMAND "Next" "View the next customer in the list"
             FETCH NEXT customer_set INTO p_customer.*
             IF status = NOTFOUND THEN
                ERROR "No more customers in this direction"
                   ATTRIBUTE(RED, REVERSE)
                FETCH LAST customer_set INTO p_customer.*
             END IF
             DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
          COMMAND "Previous" "View the previous customer in the list"
             FETCH PREVIOUS customer_set INTO p_customer.*
             IF status = NOTFOUND THEN
                ERROR "No more customers in this direction"
                   ATTRIBUTE(RED, REVERSE)
                FETCH FIRST customer_set INTO p_customer.*
             END IF
             DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
          COMMAND "First" "View the first customer in the list"
             FETCH FIRST customer_set INTO p_customer.*
             DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
          COMMAND "Last" "View the last customer in the list"
             FETCH LAST customer_set INTO p_customer.*
             DISPLAY BY NAME p_customer.*  ATTRIBUTE(CYAN)
          COMMAND "Select" "Exit BROWSE selecting the current customer"
             LET chosen = TRUE
             EXIT MENU
          COMMAND "Quit" "Quit BROWSE without selecting a customer"
             LET chosen = FALSE
             EXIT MENU
       END MENU
    END IF
    CLOSE customer_set

    IF NOT exist THEN
       CLEAR FORM
       CALL mess("No customer satisfies query", mrow)
       LET p_customer.customer_num = NULL
       RETURN (FALSE)
    END IF
    IF NOT chosen THEN
       CLEAR FORM
       LET p_customer.customer_num = NULL
       CALL mess("No selection made", mrow)
       RETURN (FALSE)
    END IF
    RETURN (TRUE)
END FUNCTION
```

```
FUNCTION update_customer()

   CALL clear_menu()
   IF p_customer.customer_num IS NULL THEN
      CALL mess("No customer has been selected; use the Find-cust option",23)
      RETURN
   END IF
   MESSAGE "Press F1 or CTRL-F for field-level help"
   DISPLAY "Press ESC to update customer data; DEL to abort" AT 1,1
   INPUT BY NAME p_customer.* WITHOUT DEFAULTS
      AFTER FIELD state
         CALL statehelp()
         DISPLAY "Press ESC to update customer data; DEL to abort", "" AT 1,1
      ON KEY (F1, CONTROL-F)
         CALL customer_help()
   END INPUT
   IF NOT int_flag THEN
      UPDATE customer SET customer.* = p_customer.*
         WHERE customer_num = p_customer.customer_num
      CALL mess("Customer data modified", 23)
   ELSE
      LET int_flag = FALSE
      SELECT * INTO p_customer.* FROM customer
         WHERE customer_num = p_customer.customer_num
      DISPLAY BY NAME p_customer.*
      ERROR "Customer update aborted" ATTRIBUTE (RED, REVERSE)
   END IF
END FUNCTION


FUNCTION delete_customer()
   DEFINE answer CHAR(1),
      num_orders INTEGER

   CALL clear_menu()
   IF p_customer.customer_num IS NULL THEN
      ERROR "No customer has been selected; use the Find-customer option"
         ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF

   SELECT COUNT(*) INTO num_orders
      FROM orders
      WHERE customer_num = p_customer.customer_num
   IF num_orders THEN
      ERROR "This customer has active orders and can not be removed"
         ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF

   PROMPT " Are you sure you want to delete this customer row? "
      FOR CHAR answer
   IF answer MATCHES "[yY]" THEN
      DELETE FROM customer
         WHERE customer_num = p_customer.customer_num
      CLEAR FORM
      CALL mess("Customer entry deleted", 23)
      LET p_customer.customer_num = NULL
   ELSE
      ERROR "Deletion aborted" ATTRIBUTE (RED, REVERSE)
   END IF
END FUNCTION
```

```
FUNCTION customer_help()
   CASE
      WHEN infield(customer_num) CALL showhelp(1001)
      WHEN infield(fname) CALL showhelp(1002)
      WHEN infield(lname) CALL showhelp(1003)
      WHEN infield(company) CALL showhelp(1004)
      WHEN infield(address1) CALL showhelp(1005)
      WHEN infield(address2) CALL showhelp(1006)
      WHEN infield(city) CALL showhelp(1007)
      WHEN infield(state) CALL showhelp(1008)
      WHEN infield(zipcode) CALL showhelp(1009)
      WHEN infield(phone) CALL showhelp(1010)
   END CASE
END FUNCTION


FUNCTION statehelp()
   DEFINE idx INTEGER

   SELECT COUNT(*) INTO idx
      FROM state
      WHERE code = p_customer.state
   IF idx = 1 THEN
      RETURN
   END IF

   DISPLAY "Move cursor using F3, F4, and arrow keys; press ESC to select state
"
      AT 1,1
   OPEN WINDOW w_state AT 8,37
      WITH FORM "state_list"
      ATTRIBUTE (BORDER, RED, FORM LINE 2)

   CALL set_count(state_cnt)
   DISPLAY ARRAY p_state TO s_state.*
   LET idx = arr_curr()

   CLOSE WINDOW w_state
   LET p_customer.state = p_state[idx].code
   DISPLAY BY NAME p_customer.state ATTRIBUTE (MAGENTA)
   RETURN
END FUNCTION
```

d4_orders.4gl

```
GLOBALS
   "d4_globals.4gl"

FUNCTION orders()

   OPEN FORM order_form FROM "orderform"
   DISPLAY FORM order_form
      ATTRIBUTE(MAGENTA)
   MENU "ORDERS"
      COMMAND "Add-order" "Enter new order to database and print invoice"
            HELP 301
         CALL add_order()
      COMMAND "Update-order" "Enter shipping or payment data" HELP 302
         CALL update_order()
      COMMAND "Find-order" "Look up and display orders" HELP 303
         CALL get_order()
      COMMAND "Delete-order" "Remove an order from the database" HELP 304
         CALL delete_order()
      COMMAND "Exit" "Return to MAIN Menu" HELP 305
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION

FUNCTION add_order()
   DEFINE pa_curr, s_curr, num_stocks INTEGER,
      file_name CHAR(20),
      query_stat INTEGER

   CALL clear_menu()
   LET query_stat = query_customer(2)
   IF query_stat IS NULL THEN
      RETURN
   END IF
   IF NOT query_stat THEN
      OPEN WINDOW cust_w AT 3,5
         WITH 19 ROWS, 72 COLUMNS
         ATTRIBUTE(BORDER, YELLOW)
      OPEN FORM o_cust FROM "custform"
      DISPLAY FORM o_cust
         ATTRIBUTE(MAGENTA)
      CALL fgl_drawbox(3,61,4,7)
      CALL fgl_drawbox(11,61,4,7)
      CALL add_customer(FALSE)
      CLOSE FORM o_cust
      CLOSE WINDOW cust_w
      IF p_customer.customer_num IS NULL THEN
         RETURN
      ELSE
         DISPLAY by name p_customer.*
      END IF
   END IF

   MESSAGE "Enter the order date, PO number and shipping instructions."
   INPUT BY NAME p_orders.order_date, p_orders.po_num, p_orders.ship_instruct
   IF int_flag THEN
      LET int_flag = FALSE
      CLEAR FORM
      ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
```

```
INPUT ARRAY p_items FROM s_items.* HELP 311
   BEFORE FIELD stock_num
      MESSAGE "Press ESC to write order"
      DISPLAY "Enter a stock number or press CTRL-B to scan stock list"
         AT 1,1
   BEFORE FIELD manu_code
      MESSAGE "Enter the code for a manufacturer"
   BEFORE FIELD quantity
      DISPLAY "" AT 1,1
      MESSAGE "Enter the item quantity"
   ON KEY (CONTROL-B)
      IF INFIELD(stock_num) OR INFIELD(manu_code) THEN
         LET pa_curr = arr_curr()
         LET s_curr = scr_line()
         CALL get_stock() RETURNING
            p_items[pa_curr].stock_num, p_items[pa_curr].manu_code,
            p_items[pa_curr].description, p_items[pa_curr].unit_price
         DISPLAY p_items[pa_curr].stock_num TO s_items[s_curr].stock_num
         DISPLAY p_items[pa_curr].manu_code TO s_items[s_curr].manu_code
         DISPLAY p_items[pa_curr].description TO s_items[s_curr].description
         DISPLAY p_items[pa_curr].unit_price TO s_items[s_curr].unit_price
         NEXT FIELD quantity
      END IF
   AFTER FIELD stock_num, manu_code
      LET pa_curr = arr_curr()
      IF p_items[pa_curr].stock_num IS NOT NULL
         AND p_items[pa_curr].manu_code IS NOT NULL
      THEN
         CALL get_item()
      END IF
   AFTER FIELD quantity
      MESSAGE ""
      LET pa_curr = arr_curr()
      IF p_items[pa_curr].unit_price IS NOT NULL
         AND p_items[pa_curr].quantity IS NOT NULL
      THEN
         CALL item_total()
      ELSE
         ERROR
         "A valid stock code, manufacturer, and quantity must all be entered
"
            ATTRIBUTE (RED, REVERSE)
         NEXT FIELD stock_num
      END IF
   AFTER INSERT, DELETE
      CALL renum_items()
      CALL order_total()
   AFTER ROW
      CALL order_total()
END INPUT

IF int_flag THEN
   LET int_flag = FALSE
   CLEAR FORM
   ERROR "Order input aborted" ATTRIBUTE (RED, REVERSE)
   RETURN
END IF
```

```
   WHENEVER ERROR CONTINUE
   BEGIN WORK
   INSERT INTO orders (order_num, order_date, customer_num,
         ship_instruct, po_num)
      VALUES (0, p_orders.order_date, p_customer.customer_num,
         p_orders.ship_instruct, p_orders.po_num)
   IF status < 0 THEN
      ROLLBACK WORK
      ERROR "Unable to complete update of orders table"
         ATTRIBUTE(RED, REVERSE, BLINK)
      RETURN
   END IF
   LET p_orders.order_num = SQLCA.SQLERRD[2]
   DISPLAY BY NAME p_orders.order_num
   IF NOT insert_items() THEN
      ROLLBACK WORK
      ERROR "Unable to insert items" ATTRIBUTE(RED, REVERSE, BLINK)
      RETURN
   END IF

   COMMIT WORK
   WHENEVER ERROR STOP
   CALL mess("Order added", 23)
   LET file_name = "inv", p_orders.order_num USING "<<<<&",".out"
   CALL invoice(file_name)
   CLEAR FORM
END FUNCTION


FUNCTION update_order()

   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION delete_order()

   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION order_total()
   DEFINE order_total MONEY(8),
      i INTEGER

   LET order_total = 0.00
   FOR i = 1 TO ARR_COUNT()
      IF p_items[i].total_price IS NOT NULL THEN
         LET order_total = order_total + p_items[i].total_price
      END IF
   END FOR
   LET order_total = 1.1 * order_total
   DISPLAY order_total TO t_price
      ATTRIBUTE(GREEN)
END FUNCTION
```

```
FUNCTION item_total()
   DEFINE pa_curr, sc_curr INTEGER

   LET pa_curr = arr_curr()
   LET sc_curr = scr_line()
   LET p_items[pa_curr].total_price =
       p_items[pa_curr].quantity * p_items[pa_curr].unit_price
   DISPLAY p_items[pa_curr].total_price TO s_items[sc_curr].total_price
END FUNCTION


FUNCTION renum_items()
   DEFINE pa_curr, pa_total, sc_curr, sc_total, k INTEGER

   LET pa_curr = arr_curr()
   LET pa_total = arr_count()
   LET sc_curr = scr_line()
   LET sc_total = 4
   FOR k = pa_curr TO pa_total
      LET p_items[k].item_num = k
      IF sc_curr <= sc_total THEN
         DISPLAY k TO s_items[sc_curr].item_num
         LET sc_curr = sc_curr + 1
      END IF
   END FOR
END FUNCTION


FUNCTION insert_items()
   DEFINE idx INTEGER

   FOR idx = 1 TO arr_count()
      IF p_items[idx].quantity != 0 THEN
         INSERT INTO items
            VALUES (p_items[idx].item_num, p_orders.order_num,
               p_items[idx].stock_num, p_items[idx].manu_code,
               p_items[idx].quantity, p_items[idx].total_price)
         IF status < 0 THEN
            RETURN (FALSE)
         END IF
      END IF
   END FOR
   RETURN (TRUE)
END FUNCTION


FUNCTION get_stock()
   DEFINE idx integer

   OPEN WINDOW stock_w AT 7, 3
      WITH FORM "stock_sel"
      ATTRIBUTE(BORDER, YELLOW)
   CALL set_count(stock_cnt)
   DISPLAY
    "Use cursor using F3, F4, and arrow keys; press ESC to select a stock item"

      AT 1,1
   DISPLAY ARRAY p_stock TO s_stock.*
   LET idx = arr_curr()
   CLOSE WINDOW stock_w
   RETURN p_stock[idx].stock_num, p_stock[idx].manu_code,
      p_stock[idx].description, p_stock[idx].unit_price
END FUNCTION
```

```
FUNCTION get_order()
   DEFINE idx, exist, chosen INTEGER,
      answer CHAR(1)

   CALL clear_menu()
   CLEAR FORM
   IF NOT query_customer(2) THEN
      RETURN
   END IF
   DECLARE order_list CURSOR FOR
      SELECT order_num, order_date, po_num, ship_instruct
         FROM orders
         WHERE customer_num = p_customer.customer_num
   LET exist = FALSE
   LET chosen = FALSE
   FOREACH order_list INTO p_orders.*
      LET exist = TRUE
      CLEAR orders.*
      FOR idx = 1 TO 4
         CLEAR s_items[idx].*
      END FOR
      DISPLAY p_orders.* TO orders.*
      DECLARE item_list CURSOR FOR
         SELECT item_num, items.stock_num, items.manu_code,
               description, quantity, unit_price, total_price
            FROM items, stock
            WHERE order_num = p_orders.order_num
               AND items.stock_num = stock.stock_num
               AND items.manu_code = stock.manu_code
            ORDER BY item_num
      LET idx = 1

      FOREACH item_list INTO p_items[idx].*
         LET idx = idx + 1
         IF idx > 10 THEN
            ERROR "More than 10 items; only 10 items displayed"
               ATTRIBUTE (RED, REVERSE)
            EXIT FOREACH
         END IF
      END FOREACH
      CALL set_count(idx - 1)
      CALL order_total()
      MESSAGE "Press ESC when you finish viewing the items"
      DISPLAY ARRAY p_items TO s_items.*
         ATTRIBUTE(CYAN)
      MESSAGE ""
      IF int_flag THEN
         LET int_flag = FALSE
         EXIT FOREACH
      END IF
      PROMPT " Enter 'y' to select this order ",
         "or RETURN to view next order: " FOR CHAR answer
      IF answer MATCHES "[yY]" THEN
         LET chosen = TRUE
         EXIT FOREACH
      END IF
   END FOREACH

   IF NOT exist THEN
      ERROR "No orders found for this customer" ATTRIBUTE (RED)
   ELSE
      IF NOT chosen THEN
         CLEAR FORM
         ERROR "No order selected for this customer" ATTRIBUTE (RED)
```

```
      END IF
   END IF
END FUNCTION


FUNCTION get_item()
   DEFINE pa_curr, sc_curr INTEGER

   LET pa_curr = arr_curr()
   LET sc_curr = scr_line()
   SELECT description, unit_price
      INTO p_items[pa_curr].description,
         p_items[pa_curr].unit_price
      FROM stock
      WHERE stock.stock_num = p_items[pa_curr].stock_num
         AND stock.manu_code = p_items[pa_curr].manu_code
   IF status THEN
      LET p_items[pa_curr].description = NULL
      LET p_items[pa_curr].unit_price = NULL
   END IF
   DISPLAY p_items[pa_curr].description, p_items[pa_curr].unit_price
      TO s_items[sc_curr].description, s_items[sc_curr].unit_price
   IF p_items[pa_curr].quantity IS NOT NULL THEN
      CALL item_total()
   END IF
END FUNCTION


FUNCTION invoice(file_name)
   DEFINE x_invoice RECORD
         order_num              LIKE orders.order_num,
         order_date             LIKE orders.order_date,
         ship_instruct          LIKE orders.ship_instruct,
         backlog                LIKE orders.backlog,
         po_num                 LIKE orders.po_num,
         ship_date              LIKE orders.ship_date,
         ship_weight            LIKE orders.ship_weight,
         ship_charge            LIKE orders.ship_charge,
         item_num               LIKE items.item_num,
         stock_num              LIKE items.stock_num,
         manu_code              LIKE items.manu_code,
         quantity               LIKE items.quantity,
         total_price            LIKE items.total_price,
         description            LIKE stock.description,
         unit_price             LIKE stock.unit_price,
         unit                   LIKE stock.unit,
         unit_descr             LIKE stock.unit_descr,
         manu_name              LIKE manufact.manu_name
      END RECORD,
      file_name CHAR(20),
      msg CHAR(40)

   DECLARE invoice_data CURSOR FOR
      SELECT o.order_num,order_date,ship_instruct,backlog,po_num,ship_date,
         ship_weight,ship_charge,
         item_num,i.stock_num,i.manu_code,quantity,total_price,
         s.description,unit_price,unit,unit_descr,
         manu_name
      FROM orders o,items i,stock s,manufact m
      WHERE
         ((o.order_num=p_orders.order_num) AND
         (i.order_num=p_orders.order_num) AND
         (i.stock_num=s.stock_num AND
         i.manu_code=s.manu_code) AND
         (i.manu_code=m.manu_code))
```

```
      ORDER BY 9
   CASE (print_option)
      WHEN "f"
         START REPORT r_invoice TO file_name
         CALL clear_menu()
         MESSAGE "Writing invoice -- please wait"
      WHEN "p"
         START REPORT r_invoice TO PRINTER
         CALL clear_menu()
         MESSAGE "Writing invoice -- please wait"
      WHEN "s"
         START REPORT r_invoice
   END CASE
   FOREACH invoice_data INTO x_invoice.*
      OUTPUT TO REPORT r_invoice (p_customer.*, x_invoice.*)
   END FOREACH
   FINISH REPORT r_invoice
   IF print_option = "f" THEN
      LET msg = "Invoice written to file ", file_name CLIPPED
      CALL mess(msg, 23)
   END IF
END FUNCTION


REPORT r_invoice (c, x)
   DEFINE c RECORD LIKE customer.*,
      x RECORD
         order_num      LIKE orders.order_num,
         order_date     LIKE orders.order_date,
         ship_instruct  LIKE orders.ship_instruct,
         backlog        LIKE orders.backlog,
         po_num         LIKE orders.po_num,
         ship_date      LIKE orders.ship_date,
         ship_weight    LIKE orders.ship_weight,
         ship_charge    LIKE orders.ship_charge,
         item_num       LIKE items.item_num,
         stock_num      LIKE items.stock_num,
         manu_code      LIKE items.manu_code,
         quantity       LIKE items.quantity,
         total_price    LIKE items.total_price,
         description    LIKE stock.description,
         unit_price     LIKE stock.unit_price,
         unit           LIKE stock.unit,
         unit_descr     LIKE stock.unit_descr,
         manu_name      LIKE manufact.manu_name
      END RECORD,
      sales_tax, calc_total MONEY(8,2)

OUTPUT
   LEFT MARGIN 0
   RIGHT MARGIN 0
   TOP MARGIN 1
   BOTTOM MARGIN 1
   PAGE LENGTH 48
```

```
FORMAT
   BEFORE GROUP OF x.order_num
      SKIP TO TOP OF PAGE
      SKIP 1 LINE
      PRINT 10 SPACES,
         "   W E S T   C O A S T   W H O L E S A L E R S ,   I N C ."
      PRINT 30 SPACES," 1400 Hanbonon Drive"
      PRINT 30 SPACES,"Menlo Park, CA  94025"
      SKIP 1 LINES
      PRINT "Bill To:", COLUMN 10,c.fname CLIPPED, " ", c.lname CLIPPED;
      PRINT COLUMN 56,"Invoice No.        ",x.order_num USING "&&&&&"
      PRINT COLUMN 10,c.company
      PRINT COLUMN 10,c.address1 CLIPPED;
      PRINT COLUMN 56,"Invoice Date: ", x.order_date
      PRINT COLUMN 10,c.address2 CLIPPED;
      PRINT COLUMN 56,"Customer No.       ", c.customer_num USING "####&"
      PRINT COLUMN 10,c.city CLIPPED,", ",c.state CLIPPED,"   ",
         c.zipcode CLIPPED;
      PRINT COLUMN 56,"PO No. ",x.po_num
      PRINT COLUMN 10,c.phone CLIPPED;
      PRINT COLUMN 56,"Backlog Status: ",x.backlog
      SKIP 1 LINES
      PRINT COLUMN 10,"Shipping Instructions: ", x.ship_instruct
      PRINT COLUMN 10,"Ship Date: ",x.ship_date USING "ddd. mmm dd, yyyy";
      PRINT "   Weight: ", x.ship_weight USING "#####&.&&"
      SKIP 1 LINES
      PRINT "----------------------------------------";
      PRINT "---------------------------------------"
      PRINT "   Stock                              Unit        ";
      PRINT "                       Item "
      PRINT " #  Num Man    Description      Qty   Cost   Unit ";
      PRINT " Unit Description      Total"
      SKIP 1 LINES
      LET calc_total = 0.00

   ON EVERY ROW
      PRINT x.item_num USING "#&","  ",
         x.stock_num USING "&&", "  ",x.manu_code;
      PRINT "  ",x.description,"  ",x.quantity USING "###&", "  ";
      PRINT x.unit_price USING "$$$&.&&","  ",x.unit, "  ",x.unit_descr,"  ";
      PRINT x.total_price USING "$$$$$$$&.&&"
      LET calc_total = calc_total + x.total_price

   AFTER GROUP OF x.order_num
      SKIP 1 LINES
      PRINT "----------------------------------------";
      PRINT "---------------------------------------"
      PRINT COLUMN 50, "        Sub-total: ",calc_total USING "$$$$$$$&.&&"
      LET sales_tax = 0.065 * calc_total
      LET x.ship_charge = 0.035 * calc_total
      PRINT COLUMN 45, "Shipping Charge (3.5%): ",
         x.ship_charge USING "$$$$$$$&.&&"
      PRINT COLUMN 50, " Sales Tax (6.5%): ",sales_tax USING "$$$$$$$&.&&"
      PRINT COLUMN 50, "                   -----------"
      LET calc_total = calc_total + x.ship_charge + sales_tax
      PRINT COLUMN 50, "            Total: ",calc_total USING "$$$$$$$&.&&"
      IF print_option = "s" THEN
         PAUSE "Type RETURN to continue"
      END IF
END REPORT
```

d4_stock.4gl

```
GLOBALS
   "d4_globals.4gl"


FUNCTION stock()
   MENU "STOCK"
      COMMAND "Add-stock" "Add new stock items to database" HELP 401
         CALL add_stock()
      COMMAND "Find-stock" "Look up specific stock item" HELP 402
         CALL query_stock()
      COMMAND "Update-stock" "Modify current stock information" HELP 403
         CALL update_stock()
      COMMAND "Delete-stock" "Remove a stock item from database" HELP 404
         CALL delete_stock()
      COMMAND "Exit" "Return to MAIN Menu" HELP 405
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION


FUNCTION add_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION query_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION update_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION delete_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION
```

d4_report.4gl

```
GLOBALS
   "d4_globals.4gl"


FUNCTION reports()
   CALL ring_menu()
   MENU "REPORTS"
      COMMAND "Labels" "Print mailing labels from customer list"
            HELP 501
         CALL print_labels()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Accounts-receivable" "Print current unpaid orders" HELP 502
         CALL print_ar()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Backlog" "Print backlogged orders" HELP 503
         CALL print_backlog()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Stock-list" "Print stock available" HELP 504
         CALL print_stock()
         CLEAR SCREEN
         CALL ring_menu()
      COMMAND "Options" "Change the report output options" HELP 505
         CALL update_options()
         CALL ring_menu()
      COMMAND "Exit" "Return to MAIN Menu" HELP 506
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION

FUNCTION print_labels()
   DEFINE where_part CHAR(200),
      query_text CHAR(250),
      msg CHAR(50),
      file_name CHAR(20)

   OPTIONS
      FORM LINE 7
   OPEN FORM customer FROM "custform"
   DISPLAY FORM customer
      ATTRIBUTE(MAGENTA)
   CALL fgl_drawbox(3,30,3,43)
   CALL fgl_drawbox(3,61,8,7)
   CALL fgl_drawbox(11,61,8,7)
   CALL clear_menu()
   DISPLAY "CUSTOMER LABELS:" AT 1,1
   MESSAGE "Use query-by-example to select customer list"
   CONSTRUCT BY NAME where_part ON customer.*
   IF int_flag THEN
      LET int_flag = FALSE
      ERROR "Label print request aborted"
      RETURN
   END IF
   MESSAGE ""
   LET query_text = "select * from customer where ", where_part CLIPPED,
      " order by zipcode"
   PREPARE label_st FROM query_text
   DECLARE label_list CURSOR FOR label_st
```

```
   CASE (print_option)
      WHEN "f"
         PROMPT " Enter file name for labels >" FOR file_name
         IF file_name IS NULL THEN
            LET file_name = "labels.out"
         END IF
         MESSAGE "Printing mailing labels to ", file_name CLIPPED,
            " -- Please wait"
         START REPORT labels_report TO file_name
      WHEN "p"
         MESSAGE "Printing mailing labels -- Please wait"
         START REPORT labels_report TO PRINTER
      WHEN "s"
         START REPORT labels_report
         CLEAR SCREEN
   END CASE
   FOREACH label_list INTO p_customer.*
      OUTPUT TO REPORT labels_report (p_customer.*)
      IF int_flag THEN
         LET int_flag = FALSE
         EXIT FOREACH
      END IF
   END FOREACH
   FINISH REPORT labels_report
   IF int_flag THEN
      LET int_flag = FALSE

      ERROR "Label printing aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   IF print_option = "f" THEN
      LET msg = "Labels printed to ", file_name CLIPPED
      CALL mess(msg, 23)
   END IF
   CLOSE FORM customer
   OPTIONS
      FORM LINE 3
END FUNCTION


REPORT labels_report (rl)
   DEFINE rl RECORD LIKE customer.*

OUTPUT
   TOP MARGIN 0
   BOTTOM MARGIN 0
   PAGE LENGTH 6

FORMAT
   ON EVERY ROW
   SKIP TO TOP OF PAGE
   PRINT rl.fname CLIPPED, 1 SPACE, rl.lname
   PRINT rl.company
   PRINT rl.address1
   IF rl.address2 IS NOT NULL THEN
      PRINT rl.address2
   END IF
   PRINT rl.city CLIPPED, ", ", rl.state, 2 SPACES, rl.zipcode
   IF print_option = "s" THEN
      PAUSE "Type RETURN to continue"
   END IF
END REPORT
```

```
FUNCTION print_ar()
   DEFINE r RECORD
         customer_num     LIKE customer.customer_num,
         fname            LIKE customer.fname,
         lname            LIKE customer.lname,
         company          LIKE customer.company,
         order_num        LIKE orders.order_num,
         order_date       LIKE orders.order_date,
         ship_date        LIKE orders.ship_date,
         paid_date        LIKE orders.paid_date,
         total_price      LIKE items.total_price
      END RECORD,
   file_name CHAR(20),
   msg CHAR(50)

   DECLARE ar_list CURSOR FOR
   SELECT customer.customer_num,fname,lname,company,
         orders.order_num,order_date,ship_date,paid_date,
         total_price
      FROM customer,orders,items
      WHERE customer.customer_num=orders.customer_num AND
         paid_date IS NULL AND
         orders.order_num=items.order_num
      ORDER BY 1,5

   CALL clear_menu()
   CASE (print_option)
      WHEN "f"
         PROMPT " Enter file name for AR Report >" FOR file_name
         IF file_name IS NULL THEN
            LET file_name = "ar.out"
         END IF
         MESSAGE "Printing AR REPORT to ", file_name CLIPPED,
            " -- Please wait"
         START REPORT ar_report TO file_name
      WHEN "p"
         MESSAGE "Printing AR REPORT -- Please wait"
         START REPORT ar_report TO PRINTER
      WHEN "s"
         START REPORT ar_report
         CLEAR SCREEN
         MESSAGE "Printing AR REPORT -- Please wait"
   END CASE

   FOREACH ar_list INTO r.*
      OUTPUT TO REPORT ar_report (r.*)
      IF int_flag THEN
         LET int_flag = FALSE
         EXIT FOREACH
      END IF
   END FOREACH
```

```
   FINISH REPORT ar_report
   IF int_flag THEN
      LET int_flag = FALSE
      ERROR "AR REPORT printing aborted" ATTRIBUTE (RED, REVERSE)
      RETURN
   END IF
   IF print_option = "f" THEN
      LET msg = "AR REPORT printed to ", file_name CLIPPED
      CALL mess(msg, 23)
   END IF
END FUNCTION
REPORT ar_report (r)
   DEFINE r RECORD
         customer_num     LIKE customer.customer_num,
         fname            LIKE customer.fname,
         lname            LIKE customer.lname,
         company          LIKE customer.company,
         order_num        LIKE orders.order_num,
         order_date       LIKE orders.order_date,
         ship_date        LIKE orders.ship_date,
         paid_date        LIKE orders.paid_date,
         total_price      LIKE items.total_price
      END RECORD,
   name_text CHAR(80)

OUTPUT
   PAGE LENGTH 22
   LEFT MARGIN 0

FORMAT
   PAGE HEADER
      PRINT 15 SPACES,"West Coast Wholesalers, Inc."
      PRINT 6 SPACES,
         "Statement of ACCOUNTS RECEIVABLE - ",
         TODAY USING "mmm dd, yyyy"
      SKIP 1 LINES
      LET name_text = r.fname CLIPPED," ",r.lname CLIPPED,"/",
         r.company CLIPPED
      PRINT 29 - length(name_text)/2 SPACES, name_text
      SKIP 1 LINES
      PRINT " Order Date   Order Number    Ship Date            Amount"
      PRINT "--------------------------------------------------------"

   BEFORE GROUP OF r.customer_num
      SKIP TO TOP OF PAGE

   AFTER GROUP OF r.order_num
      NEED 3 LINES
      PRINT " ",r.order_date,7 SPACES,r.order_num USING "###&",8 SPACES,
         r.ship_date,"  ",
         GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"

   AFTER GROUP OF r.customer_num
      PRINT 42 SPACES,"----------------"
      PRINT 42 SPACES,GROUP SUM(r.total_price) USING "$$$$,$$$,$$$.&&"
   PAGE TRAILER
      IF print_option = "s" THEN
         PAUSE "Type RETURN to continue"
      END IF
END REPORT
```

```
FUNCTION update_options()
   DEFINE po CHAR(2)

   DISPLAY "Current print option:" AT 8,25
   LET po = " ", print_option
   DISPLAY po AT 8,46 ATTRIBUTE(CYAN)
   MENU "REPORT OPTIONS"
      COMMAND "File" "Send all reports to a file"
         LET print_option = "f"
         EXIT MENU
      COMMAND "Printer" "Send all reports to the printer"
         LET print_option = "p"
         EXIT MENU
      COMMAND "Screen" "Send all reports to the terminal screen"
         LET print_option = "s"
         EXIT MENU
      COMMAND "Exit"
         EXIT MENU
   END MENU
   DISPLAY "" AT 8,1
END FUNCTION


FUNCTION print_backlog()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION


FUNCTION print_stock()
   ERROR "This option has not been implemented" ATTRIBUTE (RED)
END FUNCTION
```

d4_demo.4gl

```
FUNCTION demo()

   CALL ring_menu()
   MENU "DEMO"
      COMMAND "Menus" "Source code for MAIN Menu"
         CALL showhelp(2001)
      COMMAND "Windows" "Source code for STATE CODE Window"
         CALL showhelp(2007)
      COMMAND "Forms" "Source code for new CUSTOMER data entry"
         CALL showhelp(2006)
      COMMAND "Detail-Scrolling"
        "Source code for scrolling of new ORDER line-items"
         CALL showhelp(2003)
      COMMAND "Scroll-Cursor" "Source code for customer record BROWSE/SCROLL"
         CALL showhelp(2008)
      COMMAND "Query_language" "Source code for new order insertion using SQL"
         CALL showhelp(2004)
      COMMAND "Construct_query"
         "Source code for QUERY-BY-EXAMPLE selection and reporting"
         CALL showhelp(2002)
      COMMAND "Reports" "Source code for MAILING LABEL report"
         CALL showhelp(2005)
      COMMAND "Exit" "Return to MAIN MENU"
         CLEAR SCREEN
         EXIT MENU
   END MENU
END FUNCTION
```

helpdemo.src

```
.101
The Customer option presents you with a menu that allows you to:

      o   Add new customers to the database
      o   Locate customers in the database
      o   Update customer files
      o   Remove customers from the database

.102
The Orders option presents you with a menu that allows you to:

      o   Enter a new order and print an invoice
      o   Update an existing order
      o   Look up and display orders
      o   Remove orders from the database

.103
The Stock option presents you with a menu that allows you to:

      o   Add new items to the list of stock
      o   Look up and display stock items
      o   Modify current stock descriptions and values
      o   Remove items from the list of stock

.104
The Reports option presents you with a menu that allows you to:

      o   Select and print mailing labels sorted by zip code
      o   Print a report of current accounts receivable
      o   Print a report of backloged orders
      o   Print a list of current stock available
      o   Change the report output options

.105
The Exit option leaves the program and returns you to the operating system.

.201
The One-add option enables you to enter data on new customers to the database.
You may get assistance on what input is appropriate for each field by pressing
the function key F1 when the cursor is in the field.   When you  have  entered
all the data you want for a given customer, press ESC to enter the data in the
database. If you want to abort a given entry and not write it to the database,
press the INTERRUPT key (usually DEL or CTRL-C).

.202
The  Many-add  option  enables  you  to  enter  data  on  new customers to the
database.  You may get assistance on what input is appropriate for each  field
by  pressing  the  function key F1 when the cursor is in the field.   When you
have  entered  all  the data you want for a given customer, press ESC to enter
the data in the database.  If you want to abort a given entry and not write it
to  the database, press the INTERRUPT key (usually DEL or CTRL-C).  After each
entry, the cursor will move to the beginning of the form and await  the  entry
of the next customer.  If you have no more customers to add, press  CTRL-Z  to
return to the CUSTOMER Menu.
```

```
.203
The Find-cust option allows you to select one or more customers and to display
their  data  on the screen by using query-by-example input.  Use the RETURN or
arrow  keys to move through the form.  Enter the criteria you want the program
to use in searching for customers.  Your options include:

    o  Literal values
    o  A range of values (separated by ":")
    o  A list of values (separated by "|")
    o  Relational operators (for example ">105")
    o  Wildcards like ? and * to match single or any number of characters

.204
The Update-cust option enables you to alter data on old customers in the
database. You must first select a current customer row to deal with by using
the Find-cust option. You may get assistance on what input is appropriate for
each field by pressing the function key F1 when the cursor is in the field.
When you have altered all the data you want for a given customer, press ESC to
enter the data in the database. If you want to abort the changes and not write
them to the database, press the INTERRUPT key (usually DEL or CTRL-C).

.205
The  Delete-cust option enables you to remove customers from the database.
You must  first  select  a current customer row to deal with by using the
Find-cust option.  For your protection, you will be asked  to  confirm
that  the  record should  be  deleted.   Once  deleted,  it  cannot  be
restored.  Customers with active orders can not be deleted.

.206
The Exit option of the CUSTOMER Menu takes you back to the MAIN Menu.

.301
The  Add-order  option enables you to add a new order for an existing customer.
You must first select the desired  customer  using  query-by-example  selection
criteria.  You will  then  enter  the  order  date,  PO  number,  and  shipping
instructions.  The detail line items are then entered into a scrolling  display
array.  Up to ten items may be entered using the four line screen array.  After
the  new  order is entered, an invoice is automatically generated and displayed
on the screen.

.302
The Update-order option is currently not implemented.

.303
The  Find-order  option  enables  you  to browse through and select an existing
order.  You must first select the desired customer (or customers) who's  orders
you  wish  to  scan.  For each customer selected, each corresponding order will
be  displayed on the screen for examination. You may either select an invoice,
skip to the next invoice, or cancel processing.


.304
The Delete-order option is currently not implemented.

.305
The Exit option of the ORDER Menu returns you to the MAIN Menu.
```

```
.311
You  may enter up to ten line items into the scrolling screen array.  A number
of  standard  functions  are available for manipulating the cursor in a screen
array.

    o  F1       Insert new line in the screen array
    o  F2       Remove the current line from the screen array
    o  F3       Page down one page in the screen array
    o  F4       Page up one page in the screen array
    o  ESC      Exit input array
    o  CTRL-B   When in the Stock Number or Manufacturer Code fields,
                a window will open in the middle of the screen and
                display a scrolled list of all items in stock, identified
                by the stock number and manufacturer.  Using F3, F4, and
                the up and down arrow keys, move the cursor to the line
                that identifies the desired item and hit ESC.  The
                window will disappear and the selected information will
                automatically appear in the proper line.
    o  etc...   The arrow-keys, and the standard field editing keys
                are available

The item_total field will be displayed in reverse-video green for total
amounts over $500.

.401
The Add-stock option is currently not implemented.

.402
The Find-stock option is currently not implemented.

.403
The Update-stock option is currently not implemented.

.404
The Delete-stock option is currently not implemented.

.405
The Exit option of the STOCK Menu returns you to the MAIN Menu.

.501
The Labels option enables you to create a  list  of  mailing  labels  generated
using a query-by-example specification.  You will be prompted for  the  output
file name.

.502
The  Accounts-receivable  option enables you to create a report summarizing all
unpaid  orders in the database.  You will be prompted for the output file name.

.503
The Backlog option is currently not implemented.

.504
The Stock-list option is currently not implemented.

.505
The  Options  option  enables  you  to  change  the  destination  of any report
generated during the current session.  The default option  is  to  display  all
reports  on the terminal screen.  The other options are to print all reports to
either the printer or an operating system file.

.506
The Exit option of the REPORT Menu returns you to the MAIN Menu.
```

```
.1001
The Number field on the Customer Form contains the serial integer  assigned  to
the  customer  row  when  the  data  for the customer is first entered into the
database.  It is a unique number for each customer.  The lowest value  of  this
field is 101.

.1002
The first section following the Name label should contain the first name of the
contact person at the customer’s company.

.1003
The second section following the Name label should contain the last name of the
contact person at the customer’s company.

.1004
This field should contain the name of the customer’s company.

.1005
The  first  line  of the Address section of the form should contain the mailing
address of the company.

.1006
The  second  line  of  the Address section of the form should be used only when
there  is  not  sufficient room in the first line to contain the entire mailing
address.

.1007
The City field should contain the city name portion of the mailing  address  of
the customer.

.1008
Enter  the two-character code for the desired state.  If no code is entered, or
the entered code is not in the program’s list of valid entries, a  window  will
appear  on the screen with a scrolling list of all states and codes.  Using the
F3,  F4,  up  and  down  arrow keys, move the cursor to the line containing the
desired state.  After typing ESC, the window will disappear  and  the  selected
state code will appear in the customer entry screen.

.1009
Enter the five digit Zip Code in this field.

.1010
Enter the telephone number of the contact person at the customer’s company.
Include the Area Code and extension using the format "###-###-#### #####".

.2001
The following is the INFORMIX-4GL source for the main menu.  Note that only
the text is specified by the MENU statement; the structure and runtime menu
functions are built-in.

   OPTIONS
      HELP FILE "helpdemo"
   OPEN FORM menu_form FROM "ring_menu"
   DISPLAY FORM menu_form
   MENU "MAIN"
      COMMAND "Customer" "Enter and maintain customer data" HELP 101
         CALL customer()
         DISPLAY FORM menu_form
      COMMAND "Orders" "Enter and maintain orders" HELP 102
         CALL orders()
         DISPLAY FORM menu_form
      COMMAND "Stock" "Enter and maintain stock list" HELP 103
         CALL stock()
         DISPLAY FORM menu_form
```

```
      COMMAND "Reports" "Print reports and mailing labels" HELP 104
         CALL reports()
         DISPLAY FORM menu_form
      COMMAND "Exit" "Exit program and return to operating system" HELP 105
         CLEAR SCREEN
         EXIT PROGRAM
   END MENU

.2002
The following is the INFORMIX-4GL source code for mailing-label selection and
printing. The CONSTRUCT statement manages the query-by-example input and
builds the corresponding SQL where-clause.

   CONSTRUCT BY NAME where_part ON customer.*
   LET query_text = "select * from customer where ", where_part CLIPPED,
      " order by zipcode"
   PREPARE mail_query FROM query_text
   DECLARE label_list CURSOR FOR mail_query
   PROMPT "Enter file name for labels >" FOR file_name
   MESSAGE "Printing mailing labels to ", file_name CLIPPED," -- Please wait"
   START REPORT labels_report TO file_name
   FOREACH label_list INTO p_customer.*
      OUTPUT TO REPORT labels_report (p_customer.*)
   END FOREACH
   FINISH REPORT labels_report

See the source code option REPORT for the corresponding report routine.

.2003
The following is the INFORMIX-4GL source code for order entry using scrolled
input fields.  Only the INPUT ARRAY statement in needed to utilize the full
scrolling features.  Some additional code has been added merely to customize
the array processing to this application.

   DISPLAY "Press ESC to write order" AT 1,1
   INPUT ARRAY p_items FROM s_items.* HELP 311
      BEFORE FIELD stock_num
         MESSAGE "Enter a stock number."
      BEFORE FIELD manu_code
         MESSAGE "Enter the code for a manufacturer."
      AFTER FIELD stock_num, manu_code
         LET pa = arr_curr()
         LET sc = scr_line()
         SELECT description, unit_price
            INTO p_items[pa].description,
               p_items[pa].unit_price
            FROM stock
            WHERE stock_num = p_items[pa].stock_num AND
               stock_manu = p_items[pa].menu_code
         DISPLAY p_items[pa].description, p_items[pa].unit_price
            TO stock[sc].*
         CALL item_total()
      AFTER FIELD quantity
         CALL item_total()
      AFTER INSERT, DELETE, ROW
         CALL order_total()
   END INPUT

See the source code option QUERY-LANGUAGE for the SQL statements that
insert the order information into the database.
```

```
.2004
The following is the INFORMIX-4GL source code that uses SQL to insert the
entered order information into the database.  Note that the use of
transactions ensures that database integrity is maintained, even if an
intermediate operation fails.

   BEGIN WORK
   LET p_orders.order_num = 0
   INSERT INTO orders VALUES (p_orders.*)
   IF status < 0 THEN
      ROLLBACK WORK
      MESSAGE "Unable to complete update of orders table"
      RETURN
   END IF
   LET p_orders.order_num = SQLCA.SQLERRD[2]
   DISPLAY BY NAME p_orders.order_num
   FOR i = 1 to arr_count()
      INSERT INTO items
         VALUES (p_items[counter].item_num, p_orders.order_num,
            p_items[counter].stock_num,    p_items[counter].manu_code,
            p_items[counter].quantity,     p_items[counter].total_price)



      IF status < 0 THEN
         ROLLBACK WORK
         Message "Unable to insert items"
         RETURN FALSE
      END IF
   END FOR
   COMMIT WORK

.2005
The following is the INFORMIX-4GL source code that generates the mailing-label
report.  See the source code option CONSTRUCT for the report calling sequence.

   REPORT labels_report (rl)
      DEFINE rl RECORD LIKE customer.*
      OUTPUT
         TOP MARGIN 0
         PAGE LENGTH 6
      FORMAT
         ON EVERY ROW
            SKIP TO TOP OF PAGE
            PRINT rl.fname CLIPPED, 1 SPACE, rl.lname
            PRINT rl.company
            PRINT rl.address1
            IF rl.address2 IS NOT NULL THEN
               PRINT rl.address2
            END IF
            PRINT rl.city CLIPPED, ", ", rl.state, 2 SPACES, rl.zipcode
   END REPORT
```

```
.2006
The following is the INFORMIX-4GL source code that manages a simple form
for data entry.  Note the use of special key definitions during data entry.

   OPEN FORM cust_form FROM "customer"
   DISPLAY FORM cust_form
   MESSAGE "Press F1 or CTRL-F for field help;",
      "F2 or CTRL-Z to return to CUSTOMER Menu"
   DISPLAY "Press ESC to enter new customer data or DEL to abort entry"
   INPUT BY NAME p_customer.*
      AFTER FIELD state
         CALL statehelp()
      ON KEY (F1, CONTROL-F)
         CALL customer_help()
      ON KEY (F2, CONTROL-Z)
         CLEAR FORM
         RETURN
   END INPUT

.2007
The following is the INFORMIX-4GL source code that opens a window in the
customer entry screen, displays the list of valid state names and codes, saves
the index into the p_state array for the selected state, closes the window, and
returns the index to the calling routine.

   OPEN WINDOW w_state AT 8,40
      WITH FORM "state_list"
      ATTRIBUTE (BORDER, RED, FORM LINE 2)

   CALL set_count(state_cnt)
   DISPLAY ARRAY p_state TO s_state.*
   LET idx = arr_curr()

   CLOSE WINDOW w_state
   RETURN (idx)

.2008
The following is the INFORMIX-4GL source code that allows the user to browse
through the rows returned by a "scroll" cursor.

   DECLARE customer_set SCROLL CURSOR FOR
      SELECT * FROM customer
         ORDER BY lname
   OPEN customer_set
   FETCH FIRST customer_set INTO p_customer.*
   IF status = NOTFOUND THEN
      LET exist = FALSE
   ELSE
      LET exist = TRUE
      DISPLAY BY NAME p_customer.*
      MENU "BROWSE"
         COMMAND "Next" "View the next customer in the list"
            FETCH NEXT customer_set INTO p_customer.*
            IF status = NOTFOUND THEN
               ERROR "No more customers in this direction"
               FETCH LAST customer_set INTO p_customer.*
            END IF
            DISPLAY BY NAME p_customer.*
         COMMAND "Previous" "View the previous customer in the list"
            FETCH PREVIOUS customer_set INTO p_customer.*
            IF status = NOTFOUND THEN
               ERROR "No more customers in this direction"
               FETCH FIRST customer_set INTO p_customer.*
            END IF
            DISPLAY BY NAME p_customer.*
```

```
      COMMAND "First" "View the first customer in the list"
         FETCH FIRST customer_set INTO p_customer.*
         DISPLAY BY NAME p_customer.*
      COMMAND "Last" "View the last customer in the list"
         FETCH LAST customer_set INTO p_customer.*
         DISPLAY BY NAME p_customer.*
      COMMAND "Select" "Exit BROWSE selecting the current customer"
         LET chosen = TRUE
         EXIT MENU
      COMMAND "Quit" "Quit BROWSE without selecting a customer"
         LET chosen = FALSE
         EXIT MENU
   END MENU
END IF
CLOSE customer_set
```

Appendix

B

INFORMIX-4GL Utility Programs

This appendix describes the utility programs that are included with the **INFORMIX-4GL** software. You can invoke these utilities at the system prompt to perform the following tasks:

- The **mkmessage** utility compiles programmer-defined help messages for **INFORMIX-4GL** applications.
- The **upscol** utility enables you to establish default attributes for display fields that are linked to database columns in your screen forms. It can also establish initial default values for program variables and screen fields that you associate with columns of tables in your database.

The *mkmessage* Utility

The **mkmessage** utility converts ASCII source files that contain user messages into a format that **4GL** programs can use in on-line displays. This section describes how to use **mkmessage** with help files and with customized run-time error messages.

Programmer-Defined Help Messages

When executing an **INFORMIX-4GL** program, the user can request help whenever the program is waiting for user input. This can occur while making a menu selection, while inputting data to a form, or while responding to a prompt. You can supply help messages that are displayed whenever the user presses the Help key (specified in the OPTIONS statement). These messages can be specific to the menu option currently highlighted, or to the INPUT, INPUT ARRAY, or PROMPT statement.

Message Source Files

INFORMIX-4GL looks for the appropriate help message in the help file that you specify in an OPTIONS statement, using the HELP FILE option. You can have several help files, but only one can be in effect at a time. The structure of the message source file is as follows:

```
.num
message-text
```

where *.num* is a period, followed by an integer and *message-text* is one or more lines of characters. (Characters can include blanks.) The file can contain as many messages as you like.

Each help message should be preceded by a line with nothing on it but a period (in the first column) and a unique integer *num*. The *message-text* starts on the next line and continues until the next numbered line. Each line must end in a RETURN. All blank lines between two numbered lines are considered part of the message that belongs to the first of the two numbers.

You can use the integer *num* to identify the help message in your **INFORMIX-4GL** programs. (See the INPUT, INPUT ARRAY, MENU, and PROMPT statement descriptions in Chapter 3.)

Lines beginning with # are considered comment lines, and are ignored by **mkmessage**.

If the text of a message occupies more than 20 lines, **INFORMIX-4GL** automatically breaks the message into "pages" of 20 lines. You can change these default page breaks by entering CONTROL-L in the first column of a line in your message file to start a new page.

INFORMIX-4GL handles clearing and redisplaying the screen.

For an example of a message file, see the **helpdemo.src** file from the demonstration application on page A-58.

Creating Executable Message Files

Once you have created your message source file, you can process it for use by **INFORMIX-4GL** with this syntax:

Syntax

```
mkmessage ——————— in-file ——————— out-file ———————|
```

in-file is an ASCII source file of help messages.

out-file is the pathname of the executable output file.

Replace *in-file* and *out-file* with the names of your input and output files.

After creating an output file with the **mkmessage** utility, you should specify *out-file* in the OPTIONS statement to identify it as the current help file.

If you want to use help messages from the help file on a field-by-field basis in an INPUT or INPUT ARRAY statement, you must use the **infield( )** and **showhelp( )** library functions that are supplied with **INFORMIX-4GL**. For example, you can use these functions as the following code segment demonstrates:

```
OPTIONS
    HELP FILE "stores.hlp",
    HELP KEY F1
...
INPUT pr_fname, pr_lname, pr_phone
    FROM fname, lname, phone HELP 101
    ON KEY (F1)
        CASE
            WHEN INFIELD(lname)
                CALL showhelp(111)
            WHEN INFIELD(fname)
                CALL showhelp(112)
            WHEN INFIELD(phone)
                CALL showhelp(113)
            OTHERWISE
                CALL showhelp(101)
        END CASE
END INPUT
```

Customized Error Messages

You can also use the **mkmessage** utility to customize run-time error messages. **INFORMIX-4GL** is distributed with a file called **4glusr.msg**. This ASCII file contains some common error messages, including the messages for run-time errors that cannot be trapped by the WHENEVER ERROR statement, and messages that support the **4GL** Help menu. The **4glusr.iem** file contains the executable version of this file.

You can edit the messages in **4glusr.msg** with a text editor (for example, to make them specific to a **4GL** application, or to translate them into another language). Be sure to preserve the required numeric codes, prefixed by a period (.) to identify each message.

If you choose to modify the contents of the **4glusr.msg** message file, you must specify **4glusr.iem** in your **mkmessage** command line as the object filename:

mkmessage ***in-file*** 4glusr.iem

The executable file **4glusr.iem** is initially installed in the directory **$INFORMIX/msg**. **INFORMIX-4GL** looks for message files in one of two directories, namely **/$INFORMIXDIR/$DBLANG** or else **/$INFORMIXDIR/msg**. If **$DBLANG** is defined, **4GL** looks only in **/$INFORMIXDIR/$DBLANG**. If this is not defined, **4GL** looks only in **/$INFORMIXDIR/msg**. You must place the newly modified file **4glusr.iem** in the appropriate **/$INFORMIXDIR/msg** or **/$INFORMIXDIR/$DBLANG** directory.

The *upscol* Utility

The **upscol** utility program allows you to create and modify the **syscolval** and **syscolatt** tables, which contain default information for fields in screen forms that correspond to database columns. Chapter 5 describes these tables and their use by **INFORMIX-4GL**.

You invoke the **upscol** utility by entering the command `upscol` at the system prompt. After you select a database at the CHOOSE DATABASE screen, the following menu appears:

```
UPDATE SYSCOL: [Validate] Attributes  Exit
Update information in the data validation table.

-------------------- db-name ------------------- Press CTRL-W for Help --------
```

The options in the UPDATE SYSCOL menu are:

Validate Update the information in **syscolval**.

Attributes Update the information in **syscolatt**.

Exit Return to the operating system.

If you select either **Validate** or **Attributes**, **upscol** checks whether the corresponding table exists and, if not, asks whether you want to create it. In the text that follows, the corresponding table is called **syscol**. If you choose not to create it, enter `n`, and you will return to the UPDATE SYSCOL menu.

If the data validation table already exists, or if you enter `y` to create it, **upscol** displays the CHOOSE TABLE screen, and prompts you for the name of a table in the database. After you select a table, the CHOOSE COLUMN screen prompts you to select the name of a column whose default values you want to modify in **syscol**.

The selected table and column names appear, along with the database name, on the dividing line beneath the next menu, which is called the ACTION menu:

```
ACTION: [Add] Update  Remove  Next  Query  Table  Column  Exit
Add an entry to the data validation [or screen display attribute] table.

--------- db-name:tab-name:col-name ------------ Press CTRL-W for Help --------
```

Now **upscol** displays the first row of **syscol** that relates to the table and column in the work area beneath this menu. If no such entries exist, a message stating this appears on the Error line.

The options in the ACTION menu are:

Add Add new rows to the **syscol** table.

Update Update the currently displayed row.

Remove Remove the currently displayed row (after a prompt for verification).

Next Display the next row of **syscol**.

Query Restart the display at the first row of **syscol** for the table and column.

Table Select a new database table and column.

Column Select a new column within the chosen table.

Exit Return to the UPDATE SYSCOL menu.

Adding or Updating Under the *Validate* Option

When you select **Add** in the ACTION menu after choosing the **Validate** option in the UPDATE SYSCOL menu, the VALIDATE menu appears:

```
VALIDATE:  Autonext  Comment  Default  Include  Picture  Shift  Verify  Exit
Automatically proceed to next field when at end of current field.

--------- db-name:tab-name:col-name ------------ Press CTRL-W for Help --------
```

The options are attribute names and their selection has the following effects:

Autonext Produces a menu with three options, **Yes**, **No**, and **Exit**. **Exit** returns you to the VALIDATE menu. The default is **No**.

Comment Produces a prompt to enter a Comment line message. No quotation marks are required around the comment, but it must fit on a single screen line.

Default Produces a prompt to enter the DEFAULT attribute, formatted as described in Chapter 5. Quotation marks are required where necessary to avoid ambiguity.

Include Produces a prompt to enter the INCLUDE attribute, formatted as described in Chapter 5. Quotation marks are required where necessary to avoid ambiguity.

Picture Produces a prompt to enter the PICTURE attribute, formatted as described in Chapter 5. No quotation marks are required.

Shift Produces a menu with four options, **Up**, **Down**, **None**, and **Exit**. **Up** corresponds to the UPSHIFT attribute and **Down** to the DOWNSHIFT attribute. **Exit** returns you to the VALIDATE menu. The default is **None**.

Verify Produces a menu with three options, **Yes**, **No**, and **Exit**. **Exit** returns you to the VALIDATE menu. The default is **No**.

Exit Returns you to the ACTION menu.

The **upscol** utility adds or modifies a row of **syscolval** after you complete each of these options except **Exit**.

The **Update** option on the ACTION menu takes you immediately to the ATTRIBUTE menu or prompt corresponding to the current attribute for the current column. You can look at another attribute for the current column by using the **Next** option, start through the list again by using the **Query** option, remove the current attribute with the **Remove** option, and select a new column or table with the **Column** or **Table** options.

Adding or Updating Under the *Attribute* Option

When you select **Add** or **Update** in the ACTION menu after choosing the **Attribute** option in the UPDATE SYSCOL menu, the ATTRIBUTE menu appears:

```
ATTRIBUTE:  Blink  Color  Fmt  Left  Rev  Under  Where  Discrd_Exit  Exit_Set
Set Field blinking attribute

--------- db-name:tab-name:col-name ------------ Press CTRL-W for Help --------
```

If you are adding a new row to **syscolatt**, a default row is displayed in the work area below the menu. If you are updating an existing row of **syscolatt**, the current row appears. Since no entry is made in **syscolatt** until you select **Exit_Set**, you can alter all the attributes before deciding to modify **syscolatt** (**Exit_Set**) or to abort the changes (**Discrd_Exit**).

The options of the ATTRIBUTE menu are screen attribute names, and their selection has the following effects:

Blink Produces a menu with three options, **Yes**, **No**, and **Exit**. The default is **No**.

Color Produces a menu with the available colors (color terminals) or intensities (monochrome terminals) for display of **tab-name.col-name**. The colors displayed are those in the local **colornames** file, whose format is described in Appendix I. If no such file exists locally, **upscol** looks in **$INFORMIXDIR/incl**. If the file does not exist there, **upscol** uses the default color list (see Chapter 5). You can toggle back and forth among the colors or intensities using CONTROL-N.

Fmt	Prompts you for the format string to be used when **tab-name.col-name** is displayed.
Left	Produces a menu with three options, **Yes**, **No**, and **Exit**. **Yes** causes numeric data to be left justified within the screen field. The default is **No**.
Rev	Produces a menu with three options, **Yes**, **No**, and **Exit**. **Yes** causes the field to be displayed in reverse video. The default is **No**.
Under	Produces a menu with three options, **Yes**, **No**, and **Exit**. **Yes** causes the field to be displayed with underlining. The default is **No**.
Where	Prompts for the values and value ranges under which these attributes will apply. See Chapter 5 for allowable syntax.
Discrd_Exit	Discards the indicated changes and returns to the ACTION menu.
Exit_Set	Enters the indicated changes into the **syscolatt** table and returns to the ACTION menu.

After you complete each of these options except **Discrd_Exit**, **upscol** adds or modifies a row of **syscolatt**.

Note: *Whoever runs the* ***upscol*** *utility produces a pair of tables,* ***syscolval*** *and* ***syscolatt****, that provide default values for all the users of a database that is not MODE ANSI.*

If the current database is MODE ANSI, however, the user who runs ***upscol*** *becomes the owner of the* ***syscolatt*** *and* ***syscolval*** *tables specified at the* ***upscol*** *menus, but other users can produce their own user.* ***syscolval*** *and user.****syscolatt*** *tables. The default specifications in an* ***upscol*** *table are applied by* **INFORMIX-4GL** *only to columns of database tables that have the same owner as the* ***upscol*** *table. (For details, see the section "Default Attributes in an ANSI-Compliant Database" on page 5-72, and the notes on the INITIALIZE and VALIDATE statements in Chapter 5.)*

Appendix

C

Using C with INFORMIX-4GL

Some programming tasks may be more easily or more efficiently coded with a combination of INFORMIX-4GL code and C code. In these cases, you have two options:

- Write a **4GL** program that calls C functions
- Write a C program that calls **4GL** functions

To call either a C function or a **4GL** function, you must know about the *argument stack* mechanism (page C-2) that **4GL** uses to pass arguments between the functions and the calling code.

This appendix discusses the following general issues that relate to the application programming interface between the INFORMIX-4GL language and the C language:

- Calling a C function from a **4GL** program

  The CALL statement of **4GL** can invoke C functions that observe the calling conventions of the **4GL** argument stack (page C-8).

- Calling a **4GL** function from a C program

  For a C program to call a **4GL** function, it must include a special header file. **4GL** provides macros to initialize the argument stack and to support other requirements of the C language (page C-16).

- Decimal functions for C

  4GL provides a library of functions that facilitate the conversion of its own DECIMAL data type values to and from every data type of the C language (page C-23).

The following topics are described in this appendix:

Topic	Page
Using the Argument Stack	C-2
Passing Values Between 4GL Functions	C-3
Receiving Values from 4GL	C-4
Passing Values to 4GL	C-6
Calling a C Function from a 4GL Program	C-8
Compiling and Executing the Program	C-11
Calling a 4GL Function from a C Program	C-11
Including the fglapi.h File	C-12
Initializing the Argument Stack	C-12
Using Interrupt Signals	C-15
Compiling and Executing the C Program	C-15
Macros for Calling 4GL Functions	C-16
fgl_start( )	C-17
fgl_call( )	C-19
fgl_exitfm( )	C-21
fgl_end( )	C-22
Decimal Functions for C	C-23
deccvasc( )	C-25
dectoasc( )	C-26
deccvint( )	C-28
dectoint( )	C-29
deccvlong( )	C-30
dectolong( )	C-31
deccvflt( )	C-32
dectoflt( )	C-33
deccvdbl( )	C-34
dectodbl( )	C-35
decadd( ), decsub( ), decmul( ), and decdiv( )	C-36
deccmp( )	C-37
deccopy( )	C-38
dececvt( ) and decfcvt( )	C-39

Using the Argument Stack

Within a **4GL** program, **4GL** uses a *pushdown stack* to pass arguments and results between **4GL** functions. The caller of a function *pushes* its arguments onto the stack; the called function *pops* them off the stack to use the values. The called function pushes its return values onto the stack, and the caller pops them off to retrieve the values.

The argument stack is also used when a **4GL** program calls a C function that you have written, or when a C program calls a **4GL** function. This section describes the following:

- Passing values between **4GL** functions
- Receiving values from **4GL**
- Passing values to **4GL**

Passing Values Between 4GL Functions

Consider the following **4GL** program:

```
MAIN
    DEFINE j,k,sum,dif SMALLINT
    LET j=5
    LET k=7
    CALL sumdiff(j,k) RETURNING sum, dif
END MAIN

FUNCTION sumdiff(a,b)
    DEFINE a,b,s,d DECIMAL(16)
    LET s = a+b
    LET d = a-b
    RETURN s,d
END FUNCTION
```

When the program executes the CALL statement, **4GL** first notes the current argument stack depth. Then it pushes the function arguments onto the top of the stack in sequence from left to right (first **j** and then **k** in the example). The stack receives not only the value of an argument but its type as well (SMALLINT in the example).

The called function pops the arguments off the stack into local variables (**a** and **b**, respectively). In the example, the types of these variables (DECIMAL) are different from the types of the arguments that were pushed by the caller (SMALLINT). However, since the stack stores the type of each passed value, **4GL** is able to convert the arguments to the proper type. In this instance, **4GL** easily converts SMALLINT values to DECIMAL. Any type conversion that is supported by the LET statement is supported when popping stacked values.

The RETURN statement pushes the returned values onto the stack in sequence from left to right (first **s** and then **d** in the example). The RETURNING clause in the originating CALL statement then pops these values off the stack and into the specified variables (**sum** and **diff**, respectively), and converts the data types DECIMAL back to SMALLINT.

When **4GL** calls a function within an expression, the use of the stack is the same as in the CALL statement. The function is expected to return a single value on the stack, and **4GL** attempts to converts this value as required to evaluate the expression.

Receiving Values from 4GL

C functions or programs receive arguments from **4GL** by using the argument stack. Within the C function or program, you pop values off the stack by using *pop external functions* that are included with **4GL**. If you try to pop a value when none is on the stack, a core dump or other fatal behavior can occur.

The Pop Library Functions

This section describes the pop external functions according to the data type of the value that each pops from the argument stack.

Library Functions for Popping Numbers

The following **4GL** library functions can be called from a C function or program to pop number values from the argument stack:

```
extern void popint(int *iv)
extern void popshort(short *siv)
extern void poplong(long *liv)
extern void popflo(float *fv)
extern void popdub(double *dfv)
extern void popdec(dec_t *decv)
```

Each of these functions, like all library functions for popping values, does the following:

1. Removes one value from the argument stack.
2. Converts its data type if necessary.
3. Copies it to the designated variable.

If the value on the stack cannot be converted to the specified type, the result is undefined.

If the **popshort( )** function returns a value outside the range of -32767 to 32767, a conversion error has occurred.

The **dec_t** structure referred to by **popdec( )** is used to hold a DECIMAL value. It is discussed later in this Appendix.

Library Functions for Popping Character Strings

The following **4GL** library functions can be called to pop character values:

```
extern void popquote(char *qv, int len)
extern void popvchar(char *qv, int len)
```

Both functions copy exactly **len** bytes into the string buffer ***qv**. The **popquote( )** function pads with spaces as necessary. The **popvchar( )** does not pad to the full length. The final byte copied to the buffer is a NULL-byte to terminate the string, so the maximum string data length is **len-1**. If the stacked argument is longer than **len-1**, its trailing bytes are lost.

The **len** argument sets the maximum size of the receiving string buffer. When using **popquote( )**, you always receive exactly **len** bytes (including trailing spaces and the NULL), even if the value on the stack is an empty string. To find the true data length of a string retrieved by **popquote( )**, you must trim the trailing spaces from the popped value.

Since **4GL** can convert to CHAR any data type except TEXT or BYTE, you can use these functions to pop almost any argument.

Library Functions for Popping Time Values

The following **4GL** library functions can be called to pop DATE, DATETIME, and INTERVAL values:

```
extern void popdate(long *datv)
extern void popdtime(dtime_t *dtv, int qual)
extern void popinv(intrvl_t *iv, int qual)
```

The structure types **dtime_t** and **intrvl_t** are used to represent DATETIME and INTERVAL data in a C program. They are discussed in the *INFORMIX-ESQL/C Programmer's Manual*. The **qual** argument receives the binary representation of the DATETIME or INTERVAL qualifier. The *INFORMIX-ESQL/C Programmer's Manual* also discusses library functions for manipulating and printing DATE, DATETIME, and INTERVAL variables.

Library Functions for Popping Blobs

The following function can be called to pop a BYTE or TEXT argument:

```
extern void poplocator(loc_t **blob)
```

The structure type **loc_t** defines a BYTE or TEXT value. Its use is discussed in the *INFORMIX-OnLine Programmer's Manual*. The **poplocator( )** function is unusual in that it does not copy the passed value. The function copies only the address of the passed value (as is indicated by the double asterisk in the function prototype). Here is a C fragment showing this:

```
int get_a_text(int nargs)
{
    loc_t *theText;
    poplocator(&theText)
    ...
}
```

What **poplocator( )** stores at the address specified by its parameter is the address of the locator structure owned by the calling function. A change to the locator or the value that it describes is visible to the calling program, which is not the case with other data types.

Any BYTE or TEXT argument must be popped as BYTE or TEXT, because **4GL** provides no automatic data type conversion for these blobs.

Passing Values to 4GL

C functions or programs can pass one or more arguments to **4GL** by putting the arguments on the stack.

- When returning values to a **4GL** program from a C function, you use the *external return functions* that are provided with **4GL** to put the arguments on the stack.
- When passing values to a **4GL** function from a C program, you use the *external push functions* that are provided with **4GL** to put the arguments on the stack.

The external return functions copy their arguments to storage allocated outside the calling function. This storage is released when the returned value is popped. This makes it possible to return values from local variables of the function.

The external push functions do not make a copy of the pushed data value in allocated memory. They require a pushed value to be a variable that will be valid for the duration of the call. It is up to the calling code to dispose of the values, as necessary, after the call.

This section describes the external return and push functions.

The Return Library Functions

The following **4GL** library functions are available to return values:

```
extern void retint(int iv)
extern void retshort(short siv)
extern void retlong(long lv)
extern void retflo(float *fv)
extern void retdub(double *dfv)
extern void retdec(dec_t *decv)

extern void retquote(char *str0)
extern void retvchar(char *vc)

extern void retdate(long date)
extern void retdtime(dtime_t *dtv)

extern void retinv(intrvl_t *inv)
```

The argument of **retquote( )** is a NULL-terminated string. No library function is available for returning BYTE or TEXT values because they are passed by reference.

The C function can return data in whatever form is convenient. If conversion is possible, **4GL** converts the data type as required when popping the value. If data type conversion is not possible, an error occurs.

Note: *A C function called from* **4GL** *must always exit with the statement* **return(*n*)**, *where n is the number of return values pushed on to the stack. A function that returns nothing must exit with* **return(0)**.

The Push Library Functions

The following **4GL** library functions can be used to push number values:

```
extern void pushint(int iv)
extern void pushshort(short siv)
extern void pushlong(long liv)
extern void pushflo(float fv)
extern void pushdub(double dfv)
extern void pushdec(dec_t *decv, unsigned decp)
```

The **dec_t** structure type and the C functions for manipulating decimal data are discussed later in this Appendix. The second argument of **pushdec( )**, namely **decp**, specifies the decimal precision and scale.

For example, to give a decimal variable named **dec_var** the precision of 15 and the scale of 2, you could specify the following:

```
pushdec(dec_var, PRECMAKE(15,2));
```

Here PRECMAKE is a macro defined in the **decimal.h** file.

The following library functions can be used to push character values:

```
extern void pushquote(char *cv, int len)
extern void pushvchar(char *vcv, int len)
```

The arguments to **pushquote( )** and **pushvchar( )** are an unterminated character string and the count of characters that it contains (not including any NULL terminator).

The following library functions can be used to push DATE, DATETIME, and INTERVAL values:

```
extern void pushdate(long datv)
extern void pushdtime(dtime_t *dtv)
extern void pushinv(intrvl_t *inv)
```

This library function pushes the location of a TEXT or BYTE argument:

```
extern void pushlocator(loc_t *blob)
```

Calling a C Function from a 4GL Program

To call a C function from a **4GL** program, use the CALL statement and specify the following:

- The name of the C function
- Any arguments to pass to the C function
- Any variables to return to the **4GL** program

Note: *To run or debug a* **4GL Rapid Development System** *program that calls C functions, you must first create a customized runner. For a complete description of this process, refer to the section* *"Creating a Customized Runner" on page 1-67.*

For example, the following CALL statement calls the C function **sendmsg( )**. It passes two arguments (**chartype** and **4**, respectively) to the function and expects two arguments to be passed back (**msg_status** and **return_code**, respectively):

```
CALL sendmsg(chartype, 4) RETURNING msg_status, return_code
```

The C function receives an integer argument that specifies how many values were pushed on the argument stack (in this case, two arguments). This is the number of values to be popped off the stack in the C function. The function also needs to return values for the **msg_status** and **return_code** arguments before passing control back to the **4GL** program.

The C function should not assume it has been passed the correct number of stacked values. The C function should test its integer argument to see how many **4GL** arguments were stacked for it. (If a function is called from two or more statements in the same source module, **4GL** verifies that the same number of arguments is used in each call. A function could be called, however, from different source modules, with a different number of arguments from each module. This error, if it is an error, is not caught by **4GL**.)

This example shows a C function that requires exactly one argument:

```
int nxt_bus_day(int nargs);
{
   long theDate;
   if (nargs != 1)
   {
      fprintf(stderr,
         "nxt_bus_day: wrong number of parms (%d)\n",
         nargs );
      retdate(0L);
      return(1);
   }
   popdate(&theDate);
   switch(rdayofweek(theDate))
   {
   case 5: /* change friday -> monday */
         ++theDate;
   case 6: /* saturday -> monday*/
         ++theDate;
   default: /* (sun..thur) go to next day */
         ++theDate;
   }
   retdate(theDate); /* stack result */
   return(1) /* return count of stacked */
}
```

The function returns the date of the next business day after a given date. Since the function must receive exactly one argument, the function checks for the number of arguments passed. If the function receives a different number of arguments, it terminates the program (with an identifying message).

The C function in the next example can operate with either 1, 2, or 3 arguments. The purpose of the function is to return the index of the next occurrence of a given character in a string. The string is the first argument and is required. The second argument is the character to search for; if it is omitted, a space character is used. The third argument is an offset at which to start the search; if it is omitted, zero is used.

```
#define STSIZE 512+1
int fglindex(int nargs);
{
    char theString[STSIZE], theChar[2];
    int offset, pos;
    *theChar = ''; /* initialize defaults */
    offset = 0;
    switch(nargs)
    {
    case 3: /* fglindex(s,c,n) */
        popint(&offset);
    case 2: /* fglindex(s,c) */
        popquote(theChar,2);
    case 1: /* fglindex(s) */
        popquote(theString,STSIZE);
        break;
    default: /* zero or >3 parms, ret 0 */
        for(;nargs;--nargs) popquote(theString,STSIZE);
        retint(999);
        return(1);
    }
    if (pos = index(theString+offset,*theChar) )
        retint(offset+pos-1);
    else
        retint(0);
    return(1);
}
```

The **switch** statement is useful in popping the correct number of arguments from the stack. By arranging the valid cases in descending order, the correct number of arguments can be popped in the correct sequence with minimal coding. In this example, the C function does not terminate the **4GL** program when given an incorrect number of arguments. Instead, it disposes of all stacked arguments by popping them as character strings. Then it returns an impossible value.

Note: *A* **4GL Rapid Development System** *program that calls C functions cannot specify as an argument to the C function a 4GL program variable whose scope of reference is global.*

Compiling and Executing the Program

The version of **4GL** you are using determines how you compile and run a **4GL** program that calls C functions. If you are using the **4GL Rapid Development System**, you need to create a customized runner to handle the C functions. If you are using the **C Compiler Version**, you do not need a customized runner. For complete information on compiling and executing **4GL** programs, see Chapter 1. For information on creating a customized runner, see "RDS Programs that Call C Functions" on page 1-64.

Calling a 4GL Function from a C Program

INFORMIX-4GL provides an *application programming interface* (API) with the C language that allows you to call **4GL** functions from a C program. You can call either **4GL Rapid Development System** or **C Compiler Version** functions.

To write a C program that calls **4GL** functions, you must do the following:

1. Include the **fglapi.h** header file.
2. Execute the **fgl_start( )** macro to perform initialization tasks.
3. Execute the **fgl_call( )** macro to call each **4GL** function.
4. If the **4GL** function displays a form, execute the **fgl_exitfm( )** macro to reset your terminal for character mode.
5. At the end of the program, execute the **fgl_end( )** macro to free resources.

To pass values between the C program and **4GL** function, use the push and pop functions described in the section titled "Using the Argument Stack" on page C-2.

This section first explains how to use these features of the API with C:

- Including the **fglapi.h** file
- Initializing values
- Calling **4GL** functions
- Compiling and executing the C program
- Handling Interrupt signals

The section concludes with reference information for each API macro:

- **fgl_start( )**
- **fgl_call( )**
- **fgl_exitfm( )**
- **fgl_end( )**

Including the fglapi.h File

You must include the **fglapi.h** header file in any C program that calls **4GL** functions. This header file defines the **fgl_start( )**, **fgl_call( )**, **fgl_exitfm( )**, and **fgl_end( )** macros and is located in the **$INFORMIXDIR/incl** directory.

You can include **fglapi.h**, as demonstrated in the following example:

```
#include <fglapi.h>
o4Main()
{
    ...
}
```

Initializing the Argument Stack

Before you can call a **4GL** function in a C program, you must execute the **fgl_start( )** macro. This macro does the following:

- Initializes the argument stack so that you can pass arguments between the C program and the **4GL** functions.
- If you are using the p-code compiler, it specifies the filename (and path) of the file containing the **4GL** functions.

You can execute this macro *once* per C program.

The following example demonstrates how to call the **fgl_start( )** macro. It specifies a file named **test** as the file containing the **4GL** functions:

```
#include <fglapi.h>

o4Main()
{
fgl_start("test");
...
}
```

If you compile the **4GL** function to C code, the filename argument is optional, and is ignored if you specify it. In this case, you can call **fgl_start( )** as follows:

```
#include <fglapi.h>

o4Main()
{
fgl_start();
...
}
```

The **fgl_start( )** macro is described in detail on page C-17.

Invoking the 4GL Function

The C program must do the following to call a **4GL** function:

1. Push the argument values that the function expects onto the argument stack.
2. Use the **fgl_call( )** macro to identify the name of the **4GL** function and to tell it how many arguments to expect.

The **4GL** function must do the following to receive arguments from and to pass values back to the C program:

1. Include the appropriate arguments in the FUNCTION statement.
2. Use the DEFINE statement to define variables for all the arguments passed to the function.
3. Use the RETURN statement in the **4GL** function to return control to the C program and to list any values to pass to the calling C program.

The C program can then pop the values passed from the function off the argument stack.

For example, the C program listed on the next page calls a **4GL** function named **get_customer( )**.

The program passes one argument to the **get_customer( )** function. Then **get_customer( )** passes one argument back to the C program. The argument passed to the function is the filename and path of the demonstration database. The C program prompts the user for this filename.

The **get_customer( )** function displays a menu of the first 10 customers in the **customer** table of the specified database. The user then chooses a customer name from the menu, and the function passes the chosen name back to the C program. Finally, the C program displays the name of the customer.

```
#include <fglapi.h>
#include <stdio.h>

o4Main()
{
    char str[80];

    fgl_start("example");
    printf("enter the full path name of a STORES database: ");
    fflush(stdout);
    scanf("%s", str);
    pushquote(str, strlen(str));
    fgl_call(get_customer, 1);
    popquote(str, 80);
    printf("name entered: %s\n", str);
    fgl_end();
}
```

The logic of the **4GL** function **get_customer( )** is as follows:

```
FUNCTION get_customer(dbname)
   DEFINE dbname CHAR(30),
          cust_array ARRAY[50] of CHAR(15),
          i INT
   DATABASE dbname
   DECLARE c1 CURSOR FOR SELECT lname
      FROM customer ORDER BY lname

   LET i = 1
   FOREACH c1 INTO cust_array[i]
      LET i = i + 1
   END FOREACH

   MENU "enter name=>"
      COMMAND cust_array[1]  RETURN cust_array[1]
      COMMAND cust_array[2]  RETURN cust_array[2]
      COMMAND cust_array[3]  RETURN cust_array[3]
      COMMAND cust_array[4]  RETURN cust_array[4]
      COMMAND cust_array[5]  RETURN cust_array[5]
      COMMAND cust_array[6]  RETURN cust_array[6]
      COMMAND cust_array[7]  RETURN cust_array[7]
      COMMAND cust_array[8]  RETURN cust_array[8]
      COMMAND cust_array[9]  RETURN cust_array[9]
      COMMAND cust_array[10] RETURN cust_array[10]
   END MENU
END FUNCTION
```

Using Interrupt Signals

An **4GL** program can trap Interrupt signals by using the DEFER INTERRUPT and DEFER QUIT statements. When executing a C program that calls **4GL** functions, you must be careful how you handle Interrupts in the C program, so that you do not confuse the **4GL** signal handling with any signal handling that occurs in the C program.

The **fgl_start( )** macro defines functions to call when interrupts occur. When one of these interrupts occurs, the appropriate function clears the screen and terminates the program.

By using DEFER INTERRUPT and DEFER QUIT within a **4GL** function, you can control the processing that occurs when the interrupt is detected.

Compiling and Executing the C Program

The method by which you compile and execute a C program that calls a **4GL** function is similar to the method you use to compile and execute a **4GL** program. The following table shows the commands to compile a C program that calls **4GL** functions based on the version of **4GL** you are using:

Version of 4GL	Compilation Commands
C Compiler Version	**c4gl** command
RDS Version	**fglpc** and **cfglgo** commands

When compiling a C program that calls a **4GL** function, you must specify the -**api** option of the compilation command. Do not specify the **fgiusr.c** file on the command line unless you are calling external C functions from **4GL**.

The following examples illustrates the compilation and execution, using two source code files and one executable:

- The file **mymain.c** contains the C program.
- The file **my4gl.4gl** contains the **4GL** function.
- The file **myprog.exe** is the resulting executable.

Compiling a C Program that Calls C Compiler Version Functions

To compile a C program that calls a **C Compiler Version** function, use the **c4gl** command as shown in the following example:

```
c4gl mymain.c my4gl.4gl -o mymain.exe
./mymain.exe
```

For complete information on the **c4gl** command, see "Compiling a 4GL Module" on page 1-29.

Compiling a C Program that Calls 4GL Rapid Development System Functions

To compile a C program that calls a compiled **4GL Rapid Development System** function, use the **fglpc** and **cfglgo** commands as shown in the following example:

```
fglpc my4gl
cfglgo -api mymain.c -o mymain.exe
./mymain.exe my4gl
```

For complete information on the **fglpc** command, see "Compiling an RDS Source File" on page 1-58. For complete information on the **cfglgo** command, see "Creating a Customized Runner" on page 1-67.

Macros for Calling 4GL Functions

Four macros are provides with INFORMIX-4GL for you to use in C programs that call 4GL functions:

- **fgl_start( )**
- **fgl_call( )**
- **fgl_exitfm( )**
- **fgl_end( )**

These macros are described in the sections that follow.

fgl_start( )

The **fgl_start( )** macro initializes the **4GL** argument stack, prepares for signal handling, and, if you are using the **4GL Rapid Development System**, specifies the path of the file containing the **4GL** functions.

```
fgl_start(filename, argc, argv)
      char  *filename;
      int  argc;
      char  *argv[ ]
```

filename is the filename (and the directory path) of the file containing the **4GL** functions to call.

argc is the number of arguments on the command line, including the name of the C program currently executing.

argv is an array containing the actual arguments to the executing C program.

You can specify the *filename* by using either a quoted string or a character variable. The file extension, **.4go** or **.4gi**, is optional.

The following list describes the return codes of **fgl_start( )** and the conditions that evoke them.

0 The macro executed successfully.

< 0 The macro failed.

Usage

You must specify the **fgl_start( )** macro before using any of the following:

- The **fgl_call( )** macro
- The **4GL** pushing or popping functions

Note: *To avoid confusion, you may wish to make* **fgl_start( )** *the first function call in a C program that calls* **4GL** *functions.*

If you are using the **4GL Rapid Development System**, you must specify the *filename*. If you are using the **C Compiler Version**, the *filename* is optional. For compatibility issues, however, you may want to specify an empty string, such as " ", as a place holder for the *filename*. Specifying an empty string makes it easier to convert a **C Compiler Version** program to an **RDS Version** program.

This code example specifies **test** as the file containing the **4GL** functions:

```
#include <fglapi.h>

o4Main()
{
    ...
    fgl_start("test");
    ...
}
```

Once a **4GL** function begins execution through use of the **fgl_call( )** macro, the function has access to the arguments passed to it by **fgl_call( )** and the command line arguments passed to the calling C function itself. The arguments passed by **fgl_call( )** are accessed by the **4GL** function in the normal manner—through its argument list. The command-line arguments passed to the calling C function, however, are accessed by the **4GL** function through use of the **4GL** functions **ARG_VAL( )** and **NUM_ARGS( )**. These latter two functions operate in the normal way, as though the command-line arguments passed to the C function had been instead used as command-line arguments to execute the **4GL** MAIN function block.

fgl_call( )

The **fgl_call( )** macro calls the **4GL** function to execute. This macro passes the following arguments to the **4GL** function:

- The name of the function
- The number of arguments being passed

The **fgl_call( )** macro returns the number of arguments being passed back to the program from the function.

```
fgl_call(funcname, nparams)
    char* funcname;
    int nparams;
```

funcname is the name of the function to call.

nparams is the number of arguments you are passing to the function.

Usage

You must push on to the argument stack any values to be passed to the **4GL** function before executing the **fgl_call( )** macro. For more information on using the push functions, see the section "The Push Library Functions" on page C-7.

To read any arguments passed back to the C program from the **4GL** function, use the pop functions. For more information on using the pop functions, see the section "Receiving Values from 4GL" on page C-4.

The following C source code pushes three arguments on to the argument stack, and then calls the **out_rep1( )** function:

```
#include <fglapi.h>

o4Main()
{
    ...
        {
        fgl_start()
        ...
        pushquote(p->pw_name, strlen(p->pw_name));
        pushquote(p->pw_dir, strlen(p->pw_dir));
        pushint(p->pw_uid);
        fgl_call(out_rep1, 3);
        ...
        }
    ...
}
```

fgl_exitfm( )

The **fgl_exitfm( )** macro resets the terminal for character mode. Use this macro after calling a 4GL function that displays a form.

```
fgl_exitfm( )
```

Usage

Place this function after any **fgl_call( )** macro that causes 4GL to display one or more forms. This macro resets the terminal for character mode. If you do not execute this macro, the terminal may behave unusually, and the end user may be unable to enter any input.

The following example pushes a value on the argument stack, calls the 4GL function, pops the returned value, and then executes the **fgl_exitfm( )** macro to reset the terminal to character mode:

```
#include <fglapi.h>
#include <stdio.h>

o4Main()
{
    fgl_start()
    ...
    pushquote(str, strlen(str));
    fgl_call(get_customer, 1);
    popquote(str, 80);
    fgl_exitfm();
    ...
}
```

fgl_end( )

The **fgl_end( )** macro frees resources resulting from the execution of a C program that calls a **4GL** function.

fgl_end( )

Usage

The **fgl_end( )** macro does the following:

- Deletes any temp files created by binary large objects (blobs)
- Closes any files opened by the 4GL function
- Frees the allocated memory

Call this macro at the end of a C program that calls a **4GL** function.

The following example demonstrates popping the value returned from **4GL**, printing this value, and then freeing resources:

```
#include <fglapi.h>
#include <stdio.h>

o4Main()
{
    fgl_start()
    ...
    popquote(str, 80);
    printf("name entered: %s\n", str);
    fgl_end()
}
```

Decimal Functions for C

The data type DECIMAL is a machine-independent method for representing numbers of up to thirty-two significant digits, with or without a decimal point, and with exponents in the range -128 to +126. **4GL** provides routines that facilitate the conversion of DECIMAL-type numbers to and from every data type allowed in the C language.

DECIMAL-type numbers consist of an exponent and a mantissa (or fractional part) in base 100. In normalized form, the first digit of the mantissa must be greater than zero.

When used within a C program, DECIMAL-type numbers are stored in a C structure of the type shown below.

```
#define DECSIZE 16

struct decimal
      {
      short dec_exp;
      short dec_pos;
      short dec_ndgts;
      char  dec_dgts[DECSIZE];
      };

typedef struct decimal dec_t;
```

The **decimal** structure and the typedef **dec_t** shown above can be found in the header file **decimal.h**. Include this file in all C source files that use any of the **4GL** decimal functions:

The **decimal** structure has four parts:

dec_exp holds the exponent of the normalized DECIMAL-type number. This exponent represents a power of 100.

dec_pos holds the sign of the DECIMAL-type number (1 when the number is zero or greater; 0 when less than zero).

dec_ndgts contains the number of base-100 significant digits of the DECIMAL-type number.

dec_dgts is a character array that holds the significant digits of the normalized DECIMAL-type number (**dec_dgts**[0] != 0). Each character in the array is a one-byte binary number in base 100. The number of significant digits in **dec_dgts** is stored in **dec_ndgts**.

DECIMAL-Type Functions

All operations on DECIMAL-type numbers should take place through the functions provided in the **4GL** library, as described in the following pages. Any other operations, modifications, or analysis of DECIMAL-type numbers can produce unpredictable results.

The following C function calls are available in **4GL** to process DECIMAL-type numbers:

Function	Effect
deccvasc( )	convert C char type to DECIMAL-type
dectoasc( )	convert DECIMAL-type to C char type
deccvint( )	convert C int type to DECIMAL-type
dectoint( )	convert DECIMAL-type to C int type
deccvlong( )	convert C long type to DECIMAL-type
dectolong( )	convert DECIMAL-type to C long type
deccvflt( )	convert C float type to DECIMAL-type
dectoflt( )	convert DECIMAL-type to C float type
deccvdbl( )	convert C double type to DECIMAL-type
dectodbl( )	convert DECIMAL-type to C double type
decadd( )	add two DECIMAL numbers
decsub( )	subtract two DECIMAL numbers
decmul( )	multiply two DECIMAL numbers
decdiv( )	divide two DECIMAL numbers
deccmp( )	compare two DECIMAL numbers
deccopy( )	copy a DECIMAL number
dececvt( )	convert DECIMAL value to ASCII string
decfcvt( )	convert DECIMAL value to ASCII string

deccvasc( )

Use **deccvasc( )** to convert a value stored as a printable character in a C **char** type into a DECIMAL-type number.

```
deccvasc(cp, len, np)
    char  *cp;
    int  len;
    dec_t  *np;
```

cp points to a string that holds the value to be converted.

Leading blank spaces in the character string are ignored. The character string can have a leading plus (+) or minus (-) sign, a decimal point (.), and numbers to the right of the decimal point.

The character string can contain an exponent preceded by either *e* or *E*. The exponent value can also be preceded by a plus or minus sign.

len is the length of the string.

np is a pointer to a **dec_t** structure receiving the result of the conversion.

These are the return codes of **deccvasc( )** and the conditions that evoke them.

0	Function was successful.
-1200	Number is too large to fit into a DECIMAL-type (overflow).
-1201	Number is too small to fit into a DECIMAL-type (underflow).
-1213	String has non-numeric characters.
-1216	String has bad exponent.

Example

The following segment of code gets the character string **input** from the terminal, and converts it to **number**, a DECIMAL-type number.

```
#include <decimal.h>

char input[80];
dec_t number;
       . . .
/* get input from terminal */
getline(input);

/* convert input into decimal number */
deccvasc(input, 32, &number);
```

dectoasc( )

Use **dectoasc( )** to convert a DECIMAL-type number to an ASCII string.

```
dectoasc(np, cp, len, right)
    dec_t  *np;
    char  *cp;
    int  len;
    int  right;
```

np is a pointer to the decimal structure whose associated decimal value is to be converted to an ASCII string.

cp is a pointer to the beginning of the character buffer to hold the ASCII string.

len is the maximum length (in bytes) of the string buffer.

If the number does not fit into a character string of length *len*, then **dectoasc( )** converts the number to exponential notation. If the number still does not fit, **dectoasc( )** fills the string with asterisk symbols.

If the number is shorter than the string, it is left-justified and padded on the right with blank characters.

right is an integer indicating the number of decimal places to the right of the decimal point.

If *right* equals -1, the number of decimal places is determined by the decimal value of **np*.

Because the ASCII string returned by **dectoasc( )** is not NULL-terminated, your program must add a NULL character to the string before printing it.

The following list describes the return codes of **dectoasc( )** and the conditions that evoke them.

0 Conversion was successful.

-1 Conversion was not successful.

Example

The following segment of code accepts the character string **input** from the terminal and converts it to **number**, a DECIMAL-type number. **number** is then converted to the character string **output**, a NULL character is appended, and the string is printed.

```
#include <decimal.h>

char input[80];
char output[16];
dec_t number;
   . . .
/* get input from terminal */
getline(input);

/* convert input into decimal number */
deccvasc(input, 32, &number);

/* convert number to ASCII string */
dectoasc(&number, output, 15, 1);

/* add null character to end of string prior to printing */
output[15] = ' ';

/* print the value just entered */
printf("You just entered %s", output);
```

deccvint( )

Use **deccvint( )** to convert a C type **int** into a DECIMAL-type number.

```
deccvint(integer, np)
   int  integer;
   dec_t  *np;
```

integer is the integer that is to be converted.

np is a pointer to a **dec_t** structure receiving the result of the conversion.

The following list describes the return codes of **deccvint( )** and the conditions that evoke them.

0 Conversion was successful.

-1 Conversion was not successful.

Example

```
#include <decimal.h>

dec_t decnum;

/* convert the integer value -999
 * into a DECIMAL-type number
 */
deccvint(-999, &decnum);
```

dectoint( )

Use **dectoint( )** to convert a DECIMAL-type number into a C type **int**.

```
dectoint(np, ip)
    dec_t  *np;
    int  *ip;
```

np is a pointer to a decimal structure whose value is converted to an integer.

ip is a pointer to the integer receiving the result of the conversion.

The following list describes the return codes of **dectoint( )** and the conditions that evoke them.

0 Conversion was successful.

-1200 The magnitude of the DECIMAL-type number is greater than 32,767.

Example

```
#include <decimal.h>

dec_t mydecimal;
int myinteger;

/* convert the value in
 * mydecimal into an integer
 * and place the results in
 * the variable myinteger.
 */
dectoint(&mydecimal, &myinteger);
```

deccvlong( )

Use **deccvlong( )** to convert a C type **long** value to a DECIMAL-type number.

```
deccvlong(lng, np)
   long  lng;
   dec_t  *np;
```

lng is a pointer to a long integer.

np is a pointer to a **dec_t** structure receiving the result of the conversion.

Example

```
#include <decimal.h>

dec_t mydecimal;
long mylong;

/* Set the decimal structure
 * mydecimal to 37.
 */
deccvlong(37L, &mydecimal);
    . . .
mylong = 123456L;
/* Convert the variable mylong into
 * a DECIMAL-type number held in
 * mydecimal.
 */
deccvlong(mylong, &mydecimal);
```

dectolong( )

Use **dectolong( )** to convert a DECIMAL-type number into a C type **long**.

```
dectolong(np, lngp)
      dec_t *np;
      long  *lngp;
```

np is a pointer to a decimal structure.

lngp is a pointer to a long receiving the result of the conversion.

These are the return codes of **dectolong( )** and the conditions that evoke them.

0 Conversion was successful.

-1200 Magnitude of the DECIMAL-type number exceeds 2,147,483,647.

Example

```
#include <decimal.h>

dec_t mydecimal;
long mylong;

/* convert the DECIMAL-type value
 * held in the decimal structure
 * mydecimal to a long pointed to
 * by mylong.
 */
dectolong(&mydecimal, &mylong);
```

deccvflt( )

Use **deccvflt( )** to convert a C type **float** into a DECIMAL-type number.

```
deccvflt(flt, np)
      float flt;
      dec_t *np;
```

flt is a floating-point number.

np is a pointer to a **dec_t** structure receiving the result of the conversion.

Example

```
#include <decimal.h>

dec_t mydecimal;
float myfloat;

/* Set the decimal structure
 * myfloat to 3.14159.
 */
deccvflt(3.14159, &mydecimal);

myfloat = 123456.78;

/* Convert the variable myfloat into
 * a DECIMAL-type number held in
 * mydecimal.
 */
deccvflt(myfloat, &mydecimal);
```

dectoflt( )

Use **dectoflt( )** to convert a DECIMAL-type number into a C type **float**.

```
dectoflt(np, fltp)
	dec_t *np;
	float *fltp;
```

np is a pointer to a decimal structure.

fltp is a pointer to a floating-point number receiving the result of the conversion.

On most implementations of C, the resulting floating-point number has eight significant digits.

Example

```
#include <decimal.h>

dec_t mydecimal;
float myfloat;

/* convert the DECIMAL-type value
 * held in the decimal structure
 * mydecimal to a floating point number pointed to
 * by myfloat.
 */
dectoflt(&mydecimal, &myfloat);
```

deccvdbl( )

Use **deccvdbl( )** to convert a C type **double** into a DECIMAL-type number.

```
deccvdbl(dbl, np)
    double dbl;
    dec_t *np;
```

dbl is a double-precision floating-point number.

np is a pointer to a **dec_t** structure receiving the result of the conversion.

Example

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* Set the decimal structure
 * mydecimal to 3.14159.
 */
deccvdbl(3.14159, &mydecimal);

mydouble = 123456.78;

/* Convert the variable mydouble into
 * a DECIMAL-type number held in
 * mydecimal.
 */
deccvdbl(mydouble, &mydecimal);
```

dectodbl( )

Use **dectodbl( )** to convert a DECIMAL-type number into a C type **double**.

```
dectodbl(np, dblp)
    dec_t *np;
    double *dblp;
```

np is a pointer to a decimal structure.

dblp is a pointer to a double-precision, floating-point number that receives the result of the conversion.

The resulting double-precision value receives a total of 16 significant digits on most implementations of the C language.

Example

```
#include <decimal.h>

dec_t mydecimal;
double mydouble;

/* convert the DECIMAL-type value
 * held in the decimal structure
 * mydecimal to a double pointed to
 * by mydouble.
 */
dectodbl(&mydecimal, &mydouble);
```

decadd( ), decsub( ), decmul( ), and decdiv( )

The decimal arithmetic routines take pointers to three decimal structures as parameters. The first two decimal structures hold the operands of the arithmetic function. The third decimal structure holds the result.

```
decadd(n1, n2, result)
    dec_t  *n1;
    dec_t  *n2;
    dec_t  *result;  /* result = n1 + n2 */

decsub(n1, n2, result)
    dec_t  *n1;
    dec_t  *n2;
    dec_t  *result;  /* result = n1 - n2 */

decmul(n1, n2, result)
    dec_t  *n1;
    dec_t  *n2;
    dec_t  *result;  /* result = n1 * n2 */

decdiv(n1, n2, result)
    dec_t  *n1;
    dec_t  *n2;
    dec_t  *result;  /* result = n1 / n2 */
```

n1 is a pointer to the decimal structure of the first operand.

n2 is a pointer to the decimal structure of the second operand.

result is a pointer to the decimal structure of the result of the operation. The *result* can use the same pointer as either *n1* or *n2*.

The following list describes the return codes of the decimal arithmetic routines and the conditions that evoke them.

0 Operation was successful.

-1200 Operation resulted in overflow.

-1201 Operation resulted in underflow.

-1202 Operation attempts to divide by zero.

deccmp( )

Use **deccmp( )** to compare two DECIMAL-type numbers.

```
int deccmp(n1, n2)
        dec_t *n1;
        dec_t *n2;
```

n1 is a pointer to the decimal structure of the first number.

n2 is a pointer to the decimal structure of the second number.

The following list describes the return codes of **deccmp( )** and the conditions that evoke them.

0 The two values are the same.

-1 The first value is less than the second.

+1 The first value is greater than the second.

deccopy( )

Use **deccopy( )** to copy the value of one **dec_t** structure to another.

```
deccopy(n1, n2)
        dec_t *n1;
        dec_t *n2;
```

n1 is a pointer to the source **dec_t** structure.

n2 is a pointer to the destination **dec_t** structure.

dececvt( ) and decfcvt( )

These functions convert a DECIMAL value to an ASCII string.

```
char *dececvt(np, ndigit, decpt, sign)
        dec_t *np;
        int ndigit;
        int *decpt;
        int *sign;

char *decfcvt(np, ndigit, decpt, sign)
        dec_t *np;
        int ndigit;
        int *decpt;
        int *sign;
```

np is a pointer to a **dec_t** structure containing the value of the number that is to be converted to a string.

ndigit is, for **dececvt( )**, the length of the ASCII string; for **decfcvt( )**, it is the number of digits to the right of the decimal point.

decpt points to an integer that is the position of the decimal point relative to the beginning of the string. A negative value for ***decpt** means to the left of the returned digits.

sign is a pointer to the sign of the result. If the sign of the result is negative, ***sign** is nonzero; otherwise, the value is zero.

The **dececvt( )** function converts the decimal value pointed to by *np* into a NULL-terminated string of *ndigit* ASCII digits, and returns a pointer to the string.

The low-order digit of the DECIMAL number is rounded.

The **decfcvt( )** function is identical to **dececvt( )**, except that *ndigit* specifies the number of digits to the right of the decimal point, instead of the total number of digits.

Examples

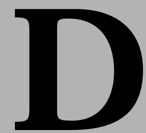
In the following example, **np** points to a **dec_t** structure containing 12345.67 and ***decpt** points to an integer containing a 5:

```
ptr = dececvt (np,4,&decpt,&sign);  = 1235
ptr = dececvt (np,10,&decpt,&sign); = 1234567000
ptr = decfcvt (np,1,&decpt,&sign);  = 123457
ptr = decfcvt (np,3,&decpt,&sign);  = 12345670
```

In this example, **np** points to a **dec_t** structure containing a 0.001234 value and ***decpt** points to an integer containing a -2 value:

```
ptr = dececvt (np,4,&decpt,&sign);  = 1234
ptr = dececvt (np,10,&decpt,&sign); = 1234000000
ptr = decfcvt (np,1,&decpt,&sign);  =
ptr = decfcvt (np,3,&decpt,&sign);  = 1
```

Appendix



Environment Variables

Various *environment variables* affect the functionality of your Informix products. You can set environment variables that identify your terminal, specify the location of your software, and define other parameters. The environment variables discussed in this appendix are grouped and listed alphabetically.

Some environment variables are required and others are optional. For example, you must set—or accept the default setting for—certain UNIX environment variables.

This appendix describes how to use the environment variables that apply to **4GL** and shows how to set them. It is divided into three main sections:

- Informix environment variables

  This section describes some standard Informix-defined environment variables that are used with **4GL**. Many of these variables are not for frequent use, but are included in case they are necessary for correct operation of **4GL** with server products. Others are for directly specifying numeric and date formatting and the locations of message files. The settings for some of these latter variables might take precedence over those for their Native Language Support (NLS) counterparts.
- NLS environment variables

  You must set some or all these variables to benefit from NLS. These might cause your product to behave differently than when their standard Informix counterparts

are set. The NLS environment is described in Appendix E, "Native Language Support Within INFORMIX-4GL."

- UNIX environment variables

  This section describes some standard UNIX system environment variables that are recognized by Informix products.

Where to Set Environment Variables

You can set Informix, NLS, and UNIX environment variables in the following ways:

- At the system prompt on the command line

  When you set an environment variable at the system prompt, you must reassign it the next time you log into the system.

- In a special shell file, as follows:

.login or **.cshrc**	for the C shell
.profile	for the Bourne shell or the Korn shell

  When you set an environment variable in your **.login**, **.cshrc**, or **.profile** file, it is assigned automatically every time you log into the system.

  Caution: *Check that you do not inadvertently set an environment variable differently in your **.login** and **.cshrc** C shell files.*

- In an environment-configuration file

  This is a common or private file where you can define all the environment variables that are used by Informix products. Using a configuration file reduces the number of environment variables that you must set at the command line or in a shell file.

An environment-configuration file can contain comment lines (preceded by #) and variable lines and their values (separated by blanks and tabs), as in the following example:

```
# This is an example of an environment-configuration file
#
# These are Informix-defined variable settings
#
DBDATE DMY4-
DBFORMAT *:.:,:DM
DBLANG german
#
# These are NLS environment variable settings
#
LANG de
```

Use the ENVIGNORE environment variable to later override one or more entries in this file. Use the following Informix **chkenv** utility to check the contents of an environment-configuration file, and return an error message if there is a bad environment-variable entry in the file or if the file is too large:

```
chkenv filename
```

The **chkenv** utility is described in Chapter 5, "SQL Utilities," in the *Informix Guide to SQL: Reference*, Version 6.0.

The common (shared) environment-configuration file resides in **$INFORMIXDIR/etc/informix.rc**. The permission for this shared file must be set to 644. A private environment-configuration file must be stored in the user's home directory as **.informix** and must be readable by the user.

Note: *The first time you set an environment variable in a shell or configuration file, before you work with your Informix product, you should log out and then log back in, "source" the file (C shell), or use "." to execute an environment-configuration file (Bourne or Korn shell). This allows the process to read your entry.*

How to Set Environment Variables

You can change default settings and add new ones by setting one or more of the environment variables recognized by your Informix product. If you are already using an Informix product, some or all the appropriate environment variables might already be set.

After one or more Informix products have been installed, enter the following command at the system prompt to view your current environment settings:

BSD UNIX: `env`

UNIX System V: `printenv`

Use standard UNIX commands to set environment variables. Depending on the type of shell you use, Figure D-1 shows how you set the fictional ABCD environment variable to *value*.

C shell:	`setenv ABCD` *`value`*
Bourne shell *or* **Korn shell:**	`ABCD=`*`value`* `export ABCD`
Korn shell:	`export ABCD=`*`value`*

Figure D-1 ***Setting environment variables in different shells***

When Bourne-shell example settings are shown in this appendix, the Korn shell (a superset of the Bourne shell) is implied as well. Note that Korn-shell syntax allows for a shortcut, as shown in Figure D-1.

Note: *The environment variables are case sensitive.*

The following diagram shows how the syntax for setting an environment variable is represented throughout this appendix. These diagrams indicate the setting for the C shell; for the Bourne or Korn shell, follow the syntax in Figure D-1.

setenv ABCD *value* ——————————————————————|

For more information on how to read syntax diagrams, see the section "Syntax Notation" in the Introduction to this manual.

To unset most of the environment variables shown in this appendix, enter the following command:

C shell: `unsetenv ABCD`

Bourne shell *or* **Korn shell:** `unset ABCD`

Default Environment Variable Settings

The following list describes the main default assumptions that are made about your environment when you use Informix products. Environment variables used to change the specific default values are shown in parentheses. Other product-specific default values are described where appropriate throughout this appendix.

- The program, compiler, or preprocessor, and any associated files and libraries of your product have been installed in the **/usr/informix** directory.
- The default **INFORMIX-OnLine Dynamic Server<Default ¶ Fo>** (**OnLine**) or **INFORMIX-SE** database server for explicit or implicit connections is indicated by an entry in the **$INFORMIXDIR/etc/sqlhosts** file. (INFORMIXSERVER)
- The default directory for message files is **$INFORMIXDIR/msg**. (DBLANG unset and LANG unset)
- If you are using **INFORMIX-SE**, the target or current database is in the current directory. (DBPATH)
- Temporary files for INFORMIX-SE are stored in the **/tmp** directory. (DBTEMP)
- The default terminal-dependent keyboard and screen capabilities are defined in the **termcap** file in the **$INFORMIXDIR/etc** directory. (INFORMIXTERM)
- For products that use an editor, the default editor is the predominant editor for the operating system, usually **vi**. (DBEDIT)
- For products that have a print capability, the program that sends files to the printer is usually:

lp	for UNIX System V
lpr	for BSD and other UNIX systems

  (DBPRINT)
- The default format for money values is `$000.00`. (DBMONEY set to *$.*)
- The default format for dates is MM/DD/YYYY. (DBDATE set to *MDY4/*)
- The field separator for unloaded data files is the vertical bar (`|=ASCII 124`). (DBDELIMITER set to |)

Rules of Precedence

When an Informix product accesses an environment variable, normally the following rules of precedence apply:

1. The highest precedence goes to the value as defined in the environment (shell).
2. The second-highest precedence goes to the value as defined in the private environment-configuration file in the user's home directory (~/**.informix**).
3. The next-highest precedence goes to the value as defined in the common environment-configuration file (**$INFORMIXDIR/etc/informix.rc**).
4. The lowest precedence goes to the default value.

If NLS is activated, there is an exception to these rules. The setting for any of the X/Open categories (LC_*) takes precedence over the setting for the LANG environment variable, no matter where they are set. For more information, see Appendix E.

The lists that follow show the most common environment variables used by Informix products. These environment variables and their uses are discussed in this appendix.

List of Environment Variables

The following tables contain alphabetical lists of the Informix, NLS, and UNIX environment variables that you can set for an Informix database server and **4GL**. These environment variables are described in this appendix on the pages listed in the last column.

INFORMIX Environment Variable	Restrictions	Page
DBANSIWARN		D-8
DBDATE		D-9
DBDELIMITER		D-11
DBEDIT		D-11
DBFORM		D-12
DBFORMAT		D-14
DBLANG		D-18
DBMONEY		D-21
DBPATH		D-23
DBPRINT		D-26

INFORMIX Environment Variable	Restrictions	Page
DBREMOTECMD	OnLine only	D-27
DBSPACETEMP	OnLine only	D-28
DBTEMP	SE only	D-29
DBUPSPACE		D-29
ENVIGNORE		D-30
INFORMIXCONRETRY		D-30
INFORMIXCONTIME		D-31
INFORMIXDIR		D-32
INFORMIXSERVER		D-33
INFORMIXSHMBASE	OnLine only	D-33
INFORMIXSTACKSIZE	OnLine only	D-34
INFORMIXTERM		D-35
ONCONFIG	OnLine only	D-36
PSORT_DBTEMP	OnLine only	D-36
PSORT_NPROCS	OnLine only	D-37
SQLEXEC		D-38
SQLRM	must be unset	D-38
SQLRMDIR	must be unset	D-39

NLS Environment Variable	Page
COLLCHAR	E-18
DBAPICODE	E-23
DBNLS	E-16
LANG	E-25
LC_COLLATE	E-27
LC_CTYPE	E-29
LC_MONETARY	E-31
LC_NUMERIC	E-35

UNIX Environment Variable	Page
PATH	D-40
TERM	D-41
TERMCAP	D-41
TERMINFO	D-42

Informix Environment Variables

This section lists alphabetically the variables that you can set when you use Informix products. It includes references to NLS environment variables that are comparable to standard Informix environment variables, where appropriate.

DBANSIWARN

The DBANSIWARN environment variable indicates that you want to check for Informix extensions to ANSI standard syntax. Unlike most environment variables, you do not need to set DBANSIWARN to a value—setting it to any value or to no value, as follows, is sufficient:

```
setenv DBANSIWARN
```

Setting the DBANSIWARN environment variable for **4GL** is functionally equivalent to including the -**ansi** flag when invoking the utility from the command line. If you set DBANSIWARN before you run **4GL**, warnings are displayed on the screen within the SQL menu.

Setting the DBANSIWARN environment variable before you compile an **4GL** program is functionally equivalent to including the -**ansi** flag in the command line. When Informix extensions to ANSI standard syntax are encountered in your program at compile time, warning messages are written to the screen.

At run time, the DBANSIWARN environment variable causes the SQL Communication Area (SQLCA) variable **sqlca.sqlwarn.sqlwarn5** to be set to `W` when a statement that is not ANSI-compliant is executed. (For more information on SQLCA, refer to the *Informix Guide to SQL: Reference*, Version 4.1.)

Once you set DBANSIWARN, Informix extension checking is automatic until you log out or unset DBANSIWARN. To turn off Informix extension checking, unset the DBANSIWARN environment variable with the following command:

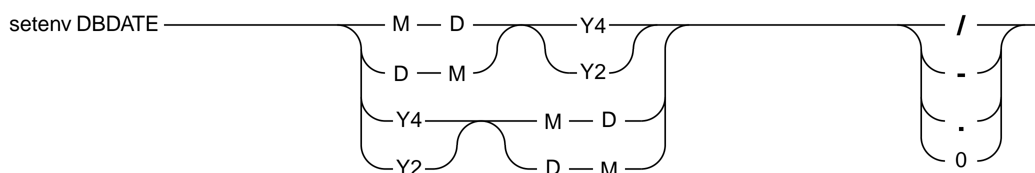
C shell: `unsetenv DBANSIWARN`

Bourne shell: `unset DBANSIWARN`

DBDATE

The DBDATE environment variable specifies the following formats for DATE values:

- The order of the month, day, and year in a date
- Whether the year should be printed with two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year



-, .	are characters that can be used as separators in a date format.
/	is the default separator for date formats.
0	is a character that indicates no separator for the date format.
D, M	are characters representing day or month, respectively, in date formats.
Y2, Y4	are characters that represent the year, and the number of digits in the year, in date formats.

The default setting for DBDATE is `MDY4/`, where `M` represents the month, `D` represents the day, `Y4` represents a four-digit year, and the slash (`/`) is a separator (for example, `12/25/1993`).

Other acceptable characters for the separator are a hyphen (-), a period (.), or a zero (0). (Use the zero to indicate no separator.)

The slash (`/`) appears if you attempt to use a character other than a hyphen, period, or zero as a separator, or if you do not include a separator character in the DBDATE definition.

You always must specify the separator character last. The number of digits you specify for the year must always follow the `Y`.

Date formatting specified in a USING clause or FORMAT attribute will override the formatting specified in DBDATE.

To make the date appear as *mmddyy*, set the DBDATE environment variable as follows:

C shell: `setenv DBDATE MDY20`

Bourne shell:
```
DBDATE=MDY20
export DBDATE
```

`MDY` represents the order of month, day, and year; `2` indicates two digits for the year; and `0` specifies no separator. As a result, the date is displayed as `122593`.

To make the date appear in European format (*dd-mm-yyyy*), set the DBDATE environment variable as follows:

C shell: `setenv DBDATE DMY4-`

Bourne shell:
```
DBDATE=DMY4-
export DBDATE
```

`DMY` represents the order of day, month, and year; `4` indicates four digits for the year; and `-` specifies a hyphen separator. As a result, the date is displayed as `25-12-1993`.

NLS

The LANG setting will specify the default value for DBDATE in an active NLS environment on HP and IBM systems. For example, a French NLS locale will establish a DBDATE default of *DMY2*. *DMY2* formats the date March 1, 1994 as *1.3.94*. On SUN systems LANG has no influence on the default for DBDATE.

DBDATE can only specify the display of month and day as numeric values, and does not support character month names and day names the way USING and FORMAT do. For this reason, changes in DBLANG and LANG do not affect the results of DBDATE on the display of DATE data.

The DBDATE variable, like DBFORMAT and DBMONEY, performs its role regardless of whether or not NLS is active (DBNLS set), and is not stored in database system tables upon database creation or considered during consistency checking. This is in contrast to LANG and the LC variables, which are only active when NLS is active, are stored with a database, and are checked for consistency.

DBDELIMITER

The DBDELIMITER environment variable specifies the field delimiter used by the **dbexport** utility in unloaded data files or with the LOAD and UNLOAD statements in **4GL**.

setenv DBDELIMITER '*delimiter*'

delimiter is the field delimiter for unloaded data files.

Any single character *except the following* is allowed:

- Hexadecimal numbers (0 through 9, a through f, A through F)
- NEWLINE or CONTROL-J
- The backslash symbol (\)

The vertical bar (`|=ASCII 124`) is the default. To change the field delimiter to a plus (+), set the DBDELIMITER environment variable as follows:

C shell:
```
setenv DBDELIMITER '+'
```

Bourne shell:
```
DBDELIMITER='+'
export DBDELIMITER
```

DBEDIT

The DBEDIT environment variable lets you name the text editor you want to use. If DBEDIT is set, the specified editor is called directly. If DBEDIT is not set, you are prompted to specify an editor as the default for the rest of the session.

setenv DBEDIT *editor*

editor is the name of the text editor you want to use.

For most systems, the default editor is **vi**. If you use another editor, be sure that it creates ASCII files. Some word processors in *document mode* introduce printer control characters that can interfere with operation of your Informix product.

To specify the EMACS text editor, set the DBEDIT environment variable as follows:

C shell:
```
setenv DBEDIT emacs
```

Bourne shell:
```
DBEDIT=emacs
export DBEDIT
```

DBFORM

The DBFORM variable specifies the subdirectory of $**INFORMIXDIR** (or full pathname) in which the menu form files for the currently active language reside. (Note that **$INFORMIXDIR** means “the name of the directory referenced by the environment variable **INFORMIXDIR**.”) Menu form files provide a set of language-translated menus to replace the standard **4GL** menus. Menu form files have the suffix **.frm**. Menu form files are included in language supplements, which contain instructions specifying where the files should be installed and what DBFORM settings to specify.

```
setenv DBFORM pathname ──────────────────────────────────────┤
```

pathname specifies the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the message files.

Usage

If DBFORM is not set, the default directory for menu form files is **$INFORMIXDIR/forms**. The files should be installed in a subdirectory under the **forms** subdirectory under **$INFORMIXDIR**. For example, French menu files could be installed in **$INFORMIXDIR/forms/french** or in **$INFORMIXDIR/forms/fr.88591**. The English language version will normally be installed in **$INFORMIXDIR/forms** or **$INFORMIXDIR/forms/english**. Non-English menu form files should not be installed in either of the locations where English files are normally found.

The following diagram illustrates the search method employed for locating message files for a particular language (where the value set in the DBFORM variable is indicated as **$DBFORM**):

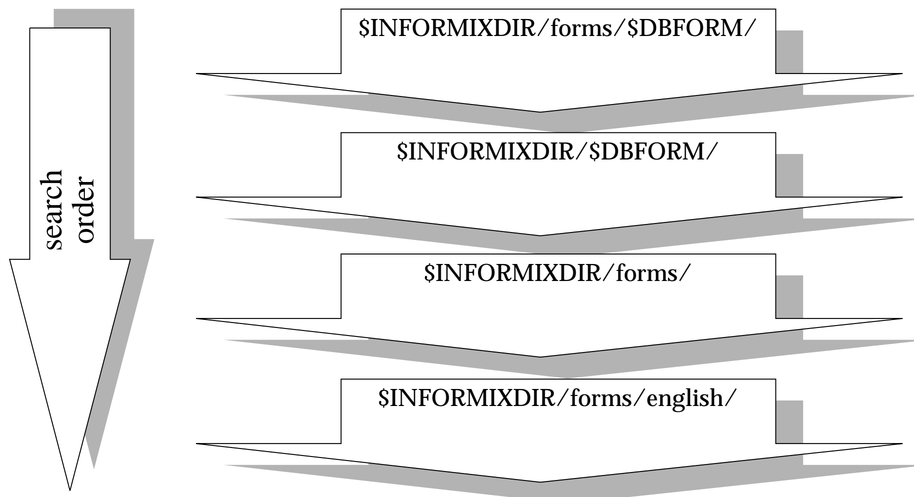


Figure D-2 Directory search order, depending on $DBFORM

NLS

If the LANG variable is set, and DBFORM is not, the search order changes to the following:

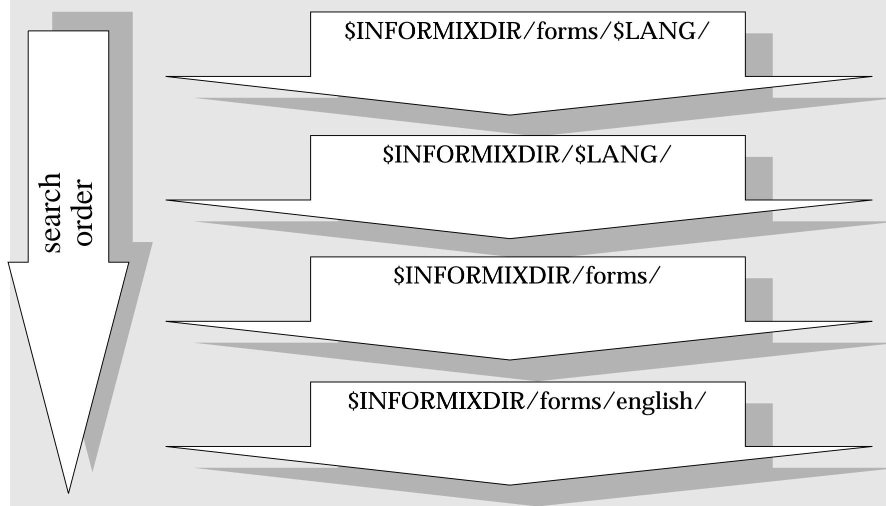


Figure D-3 Directory search order, depending on $LANG

If both DBFORM and LANG are set, LANG is ignored in establishing search order.

To specify a menu form directory, follow these steps:

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR/forms**.
2. Set the owner and group of the subdirectory to **informix** and the access permission for this directory to `755`.
3. Set the DBFORM environment variable to the new subdirectory, specifying only the subdirectory name and not the full pathname.
4. Copy the **.frm** files to the new menu form directory specified by **$INFORMIXDIR/forms/$DBFORM**. All files in the menu form directory should have the owner and group **informix** and access permission `644`.
5. Run your program or otherwise continue working with your product.

For example, you can store the set of menu form files for the French language in **$INFORMIXDIR/forms/french** as follows:

```
setenv DBFORM french
```

DBFORMAT

The Informix-defined DBFORMAT environment variable specifies the default format in which the user inputs, displays, or prints values of the following data types:

- DECIMAL
- FLOAT
- SMALLFLOAT
- INTEGER
- SMALLINT
- MONEY

The default format specified in DBFORMAT affects how numeric and monetary values are:

- Displayed and input on the screen
- Printed
- Input to and output from ASCII files using LOAD and UNLOAD

DBFORMAT is used to specify the leading and trailing currency symbols (but not their default positions within a monetary value) and the decimal and thousands separators. Note that the decimal and thousands separators defined by DBFORMAT apply to both monetary and numeric data, and override the sets of separators established by LC_MONETARY and LC_NUMERIC.

For this reason, countries which use different formatting conventions for their monetary and numeric data should use LANG, LC_MONETARY, and LC_NUMERIC, and avoid DBFORMAT.

The setting in DBFORMAT will affect the following keywords:

- USING expression
- FORMAT attribute
- PRINT statement
- LET statement, where a character string is receiving a monetary or numeric value
- DISPLAY statement

The syntax for setting DBFORMAT is as follows:

setenv DBFORMAT — *front* : *thousands* : *decimal* : *back*

front is the leading currency symbol. The *front* value is optional. The null string, represented by "*", is allowed, and means that the leading currency symbol is not applicable.

thousands is a list of one or more characters that determine the possible thousands separator. The user can use any of the specified characters as the thousands separator when inputting values. The values in the list are not separated by spaces or other characters. **4GL** uses the first value specified as the thousands separator when displaying the output value.

You can specify any characters for the thousands separator except the following:

- Digits
- <, >, |, ?, !, =, [,]

If you specify the * character, **4GL** omits the thousands separator. The *thousands* value is optional. The default value is the *. A blank space can be the thousands separator and is used for this purpose in some locales.

Note that in versions prior to 6.0, the colon symbol (:) was not allowed as a thousands separator. In version 6.0, the colon symbol is permitted, but must be preceded by a backslash (\) symbol, as in the specification **:\::.:DM**.

decimal is a list of one or more characters that determine the possible decimal separators. The user can use any of the specified

characters as the decimal separator when inputting values. **4GL** uses the first value specified as the decimal separator when displaying the output value.

You can specify any characters except the following:

- Digits
- <, >, |, ?, !, =, [,]
- Any characters specified for the *thousands* value

The *decimal* value is optional. Specification of an asterisk symbol in the decimal position will cause displayed values not to have a decimal separator.

Note that the colon symbol is permitted as a decimal separator, but must be preceded by a backslash (\) symbol in the DBFORMAT specification.

back is a value that determines the trailing currency symbol. The *back* value is optional.

You must specify all three colons in the syntax. Enclosing the DBFORMAT specification in a pair of single quotes is suggested to prevent the shell from interpreting any of the characters.

Usage

The setting in DBFORMAT directly specifies the leading and trailing currency symbol, and the numeric and decimal separators. It adds the currency symbol and changes the separators displayed on the screen in a monetary or numeric field, and in the default format of a PRINT statement. For example, if DBFORMAT is set to:

```
*:.:,:DM
```

the value 1234.56 will print or display as:

```
1234,56DM
```

DM stands for deutche marks. In the case of a screen form, values input by the user are expected to contain commas, not periods, as decimal separators if this DBFORMAT string has been specified.

The setting in DBFORMAT also affects the way format strings in the FORMAT attribute and the USING clause are interpreted. In these format strings, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by DBFORMAT. Likewise, the comma symbol (,) is a placeholder for the thousands separator specified by DBFORMAT. The dollar sign is a placeholder for the leading currency symbol. The at-sign (@) symbol

is a placeholder for the trailing currency symbol. The following table illustrates the results of different combinations of DBFORMAT setting and format string on the same value:

Value	Format String	DBFORMAT Setting	Displayed Result
1234.56	$$#,###.##	$:,:.:	$1,234.56
1234.56	$$#,###.##	:.:,:DM	1.234,56
1234.56	#,###.##@@	$:,:.:	1,234.56
1234.56	#,###.##@@	:.:,:DM	1.234,56DM

Figure D-4 ***Illustration of the results of different DBFORMAT settings and format strings***

When the user enters values, **4GL** behaves as follows:

- Disregards any currency symbols (leading or trailing) and thousands separators that the user enters.
- If a symbol appears that is defined as the decimal separator in DBFORMAT, it is interpreted in the input value as a decimal separator.

When **4GL** displays or prints values:

- The DBFORMAT-defined leading or trailing currency symbol is displayed for MONEY values.
- If a leading or trailing currency symbol is specified by the FORMAT attribute for non-MONEY data types, the symbol is displayed.
- The thousands separator does not display, unless it is included in a FORMAT attribute or USING operator.
- The decimal separator is displayed unless the decimal separator is defined as NULL (*) in DBFORMAT or the data type is integer (INT or SMALLINT).

When money values are converted to character strings using the LET statement, both the default conversion and the conversion with a USING clause will insert the DBFORMAT-defined separators and currency symbol into the created strings.

NLS

The DBFORMAT setting overrides settings in DBMONEY, LC_NUMERIC, and LC_MONETARY.

The DBFORMAT variable, like DBMONEY and DBDATE, performs its role regardless of whether or not NLS is active (DBNLS set to 1 or 2). This is in contrast to the LC variables, which are only active when NLS is active.

DBFORMAT, like DBMONEY, dictates both the numeric and monetary formats for data. In some countries, including Portugal and Italy, the correct use of decimal and thousands separators differs between numeric and monetary data. For such countries, LC_NUMERIC and LC_MONETARY provide for independently defined numeric and monetary formatting. This is in contrast to DBFORMAT and DBMONEY.

LC_NUMERIC and LC_MONETARY also can activate special logic for formatting, for purposes such as discarding the decimal portion of Italian lira. There is no fractional portion in Italian currency.

The use of LC_NUMERIC and LC_MONETARY rather than DBFORMAT is encouraged.

DBLANG

The DBLANG variable specifies the subdirectory of $**INFORMIXDIR** (or the full pathname) in which the message files for the currently active language reside. (Note that $**INFORMIXDIR** means "the name of the directory referenced by the environment variable **INFORMIXDIR**.") Message files provide a set of error messages for the engine and tools that have been translated into a national language. Message files have the suffix **.iem**.

A language supplement contains:

- Message files
- Instructions specifying where the files should be installed and what DBLANG settings to specify.

The syntax for setting DBLANG is as follows:

```
setenv DBLANG pathname
```

pathname specifies the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the message files.

Usage

If DBLANG is not set, the default directory for message files is **$INFORMIXDIR/msg**. The files should be installed in a subdirectory under the **msg** subdirectory under **$INFORMIXDIR**. For example, French message files could be installed in **$INFORMIXDIR/msg/french** or in **$INFORMIXDIR/msg/fr.88591**. The English language version will normally be installed in **$INFORMIXDIR/msg** or **$INFORMIXDIR/msg/english**. Non-English message files should not be installed in either of the locations where English files are normally found.

The following diagram illustrates the search method employed for locating message files for a particular language (where value of the variable DBLANG is designated as **$DBLANG**):

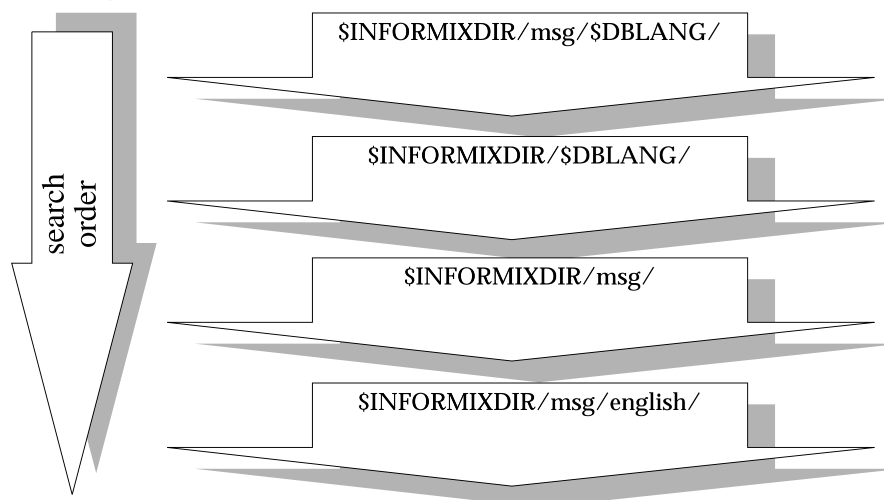


Figure D-5 Directory search order, depending on $DBLANG

NLS

If the LANG variable is set, and DBLANG is not, the search order changes to the following:

Figure D-6 Directory search order, depending on $LANG

If both DBLANG and LANG are set, LANG is ignored in establishing search order.

To specify a message directory, follow these steps:

1. Use the **mkdir** command to create the appropriate subdirectory in **$INFORMIXDIR/msg**.
2. Set the owner and group of the subdirectory to **informix** and the access permission for this directory to `755`.
3. Set the DBLANG environment variable to the new subdirectory, specifying only the subdirectory name and not the full pathname.
4. Copy the **.iem** files to the new message directory specified by **$INFORMIXDIR/msg/$DBLANG**. All files in the message directory should have the owner and group **informix** and access permission `644`.
5. Run your program or otherwise continue working with your product.

For example, you can store the set of message files for the French language in **$INFORMIXDIR/msg/french** as follows:

```
setenv DBLANG french
```

NLS

Informix tools do not support the XPG3 category LC_MESSAGES because the use of LC_MESSAGES requires storage of messages in system directories, which is less desirable than using the standard Informix message directory method.

DBMONEY

The DBMONEY environment variable specifies the display format for MONEY values.

The syntax is as follows:

setenv DBMONEY $ front . , back

$ is the default symbol that precedes the MONEY value.

. is the default decimal separator symbol that separates the integral from the fractional part of the MONEY value.

, is an alternative decimal separator symbol that separates the integral from the fractional part of the MONEY value.

back represents the optional trailing currency symbol that follows the MONEY value. The *back* symbol can be up to seven characters long and can contain any character except a comma or a period.

front is the alternative leading currency symbol that precedes the MONEY value instead of $. The *front* symbol can be up to seven characters long and can contain any character except a comma or a period.

The default setting for DBMONEY is: **$.**

where a dollar sign ($) precedes the MONEY value, a period (.) separates the integral from the fractional part of the MONEY value, and no *back* symbol appears. For example, `100.50` is formatted as `$100.50`.

Suppose you want to represent MONEY values in deutsche marks, which use *DM* as the currency symbol and a comma as the decimal separator. Set the DBMONEY environment variable as follows:

C shell: `setenv DM,`

Bourne shell:
```
DBMONEY=DM,
export DBMONEY
```

Here, `DM` is the currency symbol preceding the MONEY value, and a comma separates the integral from the fractional part of the MONEY value. As a result, the amount `100.50` is displayed as `DM100,50`.

Whenever you supply a *back* symbol, you must also supply the *front* symbol and the decimal separator (a comma or period). Similarly, if you change the value separator from a period to a comma, you must also supply the *front* symbol.

Selecting the period as a decimal separator dictates the use of the comma as a thousands separator for monetary and numeric values. Selecting the comma as a decimal separator dictates the use of the period as the thousands separator.

NLS

DBMONEY represents syntax from older versions of the product set. It is recommended that you use the LC_MONETARY and LANG, or DBFORMAT, environment variables for specifying monetary format. DBMONEY has been retained only for compatibility with older versions.

The DBMONEY variable, like DBFORMAT and DBDATE, performs its role regardless of whether or not NLS is active (DBNLS set to 1 or 2). This is in contrast to LANG and the LC variables, which are only active when NLS is active.

The contents of a *declared* DBMONEY environment variable take precedence over the contents of the LC_MONETARY variable that can be set for NLS. However, the LC_MONETARY setting takes precedence over the *default* DBMONEY format.

DBMONEY, like DBFORMAT, dictates both the numeric and monetary formats for data. In some countries, including Portugal and Italy, the correct use of decimal and thousands separators differs between numeric and monetary data. For such countries, LC_NUMERIC and LC_MONETARY provide for independently defined numeric and monetary formatting. This is in contrast to DBMONEY and DBFORMAT.

LC_NUMERIC and LC_MONETARY also can activate special logic for formatting, for purposes such as discarding the decimal portion of Italian lira. There is no fractional portion in Italian currency.

The use of LC_NUMERIC and LC_MONETARY rather than DBFORMAT or DBMONEY is encouraged. The use of DBFORMAT rather than DBMONEY is encouraged, if LC_ variables cannot be used.

See the discussion of LC_MONETARY in Appendix E, "Native Language Support Within INFORMIX-4GL."

DBPATH

Use DBPATH to identify the database servers that contain databases (if you are using the **OnLine** server), or the directories and/or database servers that contain databases (if you are using **INFORMIX-SE**). The DBPATH environment variable also specifies a list of directories (in addition to the current directory) in which **4GL** looks for command scripts (**.sql** files).

The CONNECT, DATABASE, START DATABASE, and DROP DATABASE statements use DBPATH to locate the database under two conditions:

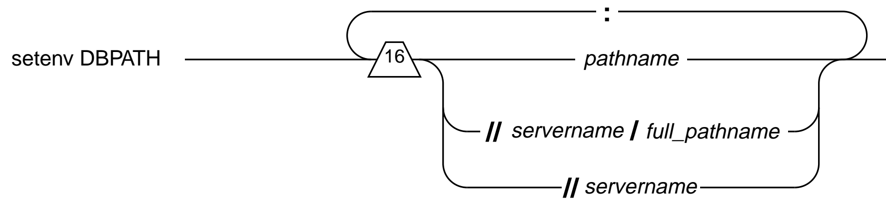
- If the location of a database is not explicitly stated and if the database cannot be located in the default server

  or

- For **INFORMIX-SE**, the default directory

The CREATE DATABASE statement does not use DBPATH.

You can add a new DBPATH entry to existing entries. To do so, use the $ format described for the UNIX environment variable PATH, described on page D-40.



pathname	is a valid relative pathname for a directory in which **.sql** files are stored or in which **INFORMIX-SE** databases are stored.
full_pathname	is a valid full pathname, starting with root, for a directory in which **.sql** files are stored or in which **INFORMIX-SE** databases are stored.
servername	is the name of an **OnLine** or **INFORMIX-SE** database server on which databases are stored. You cannot reference database files with a servername.

DBPATH can contain up to 16 entries. Each entry (*pathname*, or *servername*, or *servername* and *full_pathname*) must be less than 128 characters long. In addition, the maximum length of DBPATH depends on the hardware platform on which you are setting DBPATH.

When you access a database using the CONNECT, DATABASE, START DATABASE, or DROP DATABASE statement, the search for the database is done first in the directory or database server specified in the statement. If no database server is specified, the default database server as set in the INFORMIXSERVER environment variable is used. For **INFORMIX-SE**, if no directory is specified in the statement, the default directory is searched for the database. (The default directory is the current working directory if the database server is on the local machine, or your login directory if the database server is on a remote machine.) If a directory is specified but is not a full path, the directory is considered to be relative to the default directory.

If the database is not located during the initial search, and if DBPATH is set, the database servers or directories in DBPATH are searched for the indicated database. The entries to DBPATH are considered in order.

Searching Local Directories

Use a pathname without a database server name to have the database server search for databases or **.sql** scripts on your local machine. If you are using **4GL** with **INFORMIX-SE**, you can search for a database and **.sql** scripts; with **OnLine**, you can look only for **.sql** scripts.

For example, the following DBPATH setting causes **4GL** to search for the database files in your current directory and then in Joachim's and Sonja's directories on the local machine:

```
setenv DBPATH /usr/joachim:/usr/sonja
```

As shown in the previous example, if the pathname specifies a directory name but not a database server name, the directory is sought on the machine running the default database server as specified by the INFORMIXSERVER environment variable. (See page D-33.) For instance, with this example, if INFORMIXSERVER is set to **quality**, the DBPATH value is *interpreted* as follows, where the double slash precedes the database server name:

```
setenv DBPATH //quality/usr/joachim://quality/usr/sonja
```

Searching Networked Machines for Databases

If you are using more than one database server, you can set DBPATH to explicitly contain the database server and/or directory names that you want to be searched for databases. For example, if INFORMIXSERVER is set to **quality** but you also want to search the **marketing** database server for **/usr/joachim**, set DBPATH as follows:

```
setenv DBPATH //marketing/usr/joachim:/usr/sonja
```

Specifying a Servername

You can set DBPATH to contain only database server names. This allows you to locate only databases and not locate command files.

The **OnLine** or **SE** administrator must include each database server mentioned by DBPATH in the **$INFORMIXDIR/etc/sqlhosts** file. For information on communication-configuration files and dbservernames, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide,* Version 6.0, or *INFORMIX-SE Administrator's Guide*, Version 6.0.

For example, if INFORMIXSERVER is set to **quality**, you can search for an **OnLine** database first on the **quality** database server and then on the **marketing** database server by setting DBPATH as follows:

```
setenv DBPATH //marketing
```

For **INFORMIX-SE**, you can set DBPATH to contain just the database server names (and no directory names) if you want to locate only databases and not command scripts:

- If you specify a local **SE** database server, the current working directory is searched for databases.
- If you specify a remote **SE** database server, the search for databases is done in the login directory of the user on the machine where the database server is running.

DBPRINT

The DBPRINT environment variable specifies the print program that you want to use.

setenv DBPRINT *program*

program names any command, shell script, or UNIX utility that handles standard ASCII input.

The default program is as follows:

- For most BSD UNIX systems, the default program is **lpr**.
- For UNIX System V, the default program is usually **lp**.

Set the DBPRINT environment variable as follows to specify the **myprint** print program:

C shell:
```
setenv DBPRINT myprint
```

Bourne shell:
```
DBPRINT=myprint
export DBPRINT
```

DBREMOTECMD

You can set the DBREMOTECMD environment variable to override the default remote shell used when you perform remote tape operations with **OnLine**. Set it using either a simple command or the full pathname. If you use the full pathname, the database server searches your PATH for the specified command.

```
setenv DBREMOTECMD ─────────┬─ command ──┬──────┤
                            └─ pathname ─┘
```

command is the command to override the default remote shell.

pathname is the pathname to override the default remote shell.

Informix highly recommends the full pathname syntax on the interactive UNIX platform to avoid problems with like-named programs in other directories and possible confusion with the *restricted shell* (**/usr/bin/rsh**).

Set the DBREMOTECMD environment variable as follows for a simple command name:

C shell:

```
setenv DBREMOTECMD rcmd
```

Bourne shell:

```
DBREMOTECMD=rcmd
export DBREMOTECMD
```

Set the DBREMOTECMD environment variable as follows to specify the full pathname:

C shell:

```
setenv DBREMOTECMD /usr/bin/remsh
```

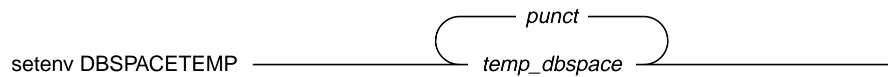
Bourne shell:

```
DBREMOTECMD=/usr/bin/remsh
export DBREMOTECMD
```

For more information on DBREMOTECMD, see the discussion in the *INFORMIX-OnLine Dynamic Server Archive and Backup Guide,* Version 6.0, about using remote tape devices with **OnLine** for archives, restores, and logical log backups.

DBSPACETEMP

If you are using **OnLine**, you can set your DBSPACETEMP environment variable to specify the dbspace to be used for building all temporary tables and for holding temporary files used for sorting in OnLine. This spreads temporary space across any number of disks.



punct can be either colons or commas.

temp_dbspace is a list of valid existing temporary dbspaces.

You can set the DBSPACETEMP environment variable to override the default dbspaces used for temporary tables and sorting space specified in the DBSPACETEMP configuration parameter in the **OnLine** configuration file. For example, you might set DBSPACETEMP as follows:

C shell:
```
setenv DBSPACETEMP sorttmp1: sorttmp2: sorttmp3
```

Bourne shell:
```
DBSPACETEMP=sorttmp1:sorttmp2:sorttmp3
export DBSPACETEMP
```

Separate the dbspace entries with either colons or commas. The number of dbspaces is limited by the maximum size of the environment variable, as defined by the UNIX shell.

The default, if left unspecified, becomes the root dbspace. **OnLine** does not create the dbspace named by the environment variable if it does not exist.

For the creation of temporary tables, if neither DBSPACETEMP nor the DBSPACETEMP parameter in the **onconfig** file is set, **OnLine** creates the temporary tables in the dbspace where the database was created.

For sorting space, **OnLine** uses the following disk space for writing temporary information, in the following order:

1. The operating system directory or directories specified by the environment variable PSORT_DBTEMP, if set.
2. The dbspace or dbspaces specified by the environment variable DBSPACETEMP, if set.
3. The dbspace or dbspaces specified by the **onconfig** parameter DBSPACETEMP.
4. The operating system file space in **/tmp.**

DBTEMP

Set the DBTEMP environment variable to specify the full pathname of the directory into which you want **INFORMIX-SE** to place its temporary files. You need not set DBTEMP if the default, **/tmp**, is satisfactory.

```
setenv DBTEMP pathname
```

pathname is the full pathname of the directory for temporary files.

Set the DBTEMP environment variable as follows to specify the pathname **usr/magda/mytemp**:

C shell:
```
setenv DBTEMP usr/magda/mytemp
```

Bourne shell:
```
DBTEMP=usr/magda/mytemp
export DBTEMP
```

For the creation of temporary tables, if DBTEMP is not set, the temporary tables are created in the directory of the database (that is, the **.dbs** directory).

DBUPSPACE

The DBUPSPACE environment variable lets you specify and thus constrain the amount of system disk space that the UPDATE STATISTICS statement can use when trying to simultaneously construct multiple column distributions.

```
setenv DBUPSPACE value
```

value represents a disk space amount in kilobytes.

For example, if DBUPSPACE is set to 2500 (kilobytes) as follows,

C shell:
```
setenv DBUPSPACE 2500
```

Bourne shell:
```
DBUPSPACE=2500
export DBUPSPACE
```

then no more than 2.5 megabytes of disk space are to be used to accomplish sorting during the execution of an UPDATE STATISTICS statement. If a table requires 5 megabytes of disk space for sorting, then UPDATE STATISTICS accomplishes the task in two passes; the distributions for one half of the columns are constructed with each pass.

If you try to set DBUPSPACE to any value less than 1024 kilobytes, it is automatically set to 1024 kilobytes, but no error message is returned. If this value is not large enough to allow more than one distribution to be constructed at a time, at least one distribution is done, even if the amount of disk space required for the one is greater than specified in DBUPSPACE.

ENVIGNORE

Use the ENVIGNORE environment variable to deactivate specified environment variable entries in the common (shared) and private environment-configuration files.

```
                                   ┌────── : ──────┐
setenv ENVIGNORE ──────────────────┴─── variable ──┴──────────────┤
```

variable is the list of environment variables that you want to deactivate.

For example, to ignore the DBPATH and DBMONEY entries in the environment-configuration files, specify the following command:

C shell:
```
setenv ENVIGNORE DBPATH:DBMONEY
```

Bourne shell:
```
ENVIGNORE=DBPATH:DBMONEY
export ENVIGNORE
```

The common environment-configuration file is stored in **$INFORMIXDIR/etc/informix.rc**. The private environment-configuration file is stored in the user's home directory as **.informix**. See "Where to Set Environment Variables" on page D-2 for information on creating or modifying an environment-configuration file.

Note: *ENVIGNORE cannot be set in an environment-configuration file.*

INFORMIXCONRETRY

The INFORMIXCONRETRY environment variable lets you specify the maximum number of *additional* connection attempts that should be made to each server by the client during the time limit specified by the INFORMIXCONTIME environment variable.

Set the INFORMIXCONRETRY environment variable as follows:

```
setenv INFORMIXCONRETRY value ────────────────────────────────────┤
```

value represents the number of connection attempts to each server.

For example, set INFORMIXCONRETRY to 3 additional connection attempts (after the initial attempt) as follows:

C shell:
```
setenv INFORMIXCONRETRY 3
```

Bourne shell:
```
INFORMIXCONRETRY=3
export INFORMIXCONRETRY
```

The default value for INFORMIXCONRETRY is one retry after the initial connection attempt. The INFORMIXCONTIME setting, described below, takes precedence over the INFORMIXCONRETRY setting.

INFORMIXCONTIME

The INFORMIXCONTIME environment variable lets you specify the minimum time limit, in seconds, for the SQL statement CONNECT to attempt to connect to a server before it returns an error.

You might encounter connection difficulties related to system or network load problems. For instance, if the server is busy establishing new SQL client threads, some of the clients might fail because the server can not issue a network function call fast enough. The INFORMIXCONTIME and INFORMIXCONRETRY environment variables let you configure your client-side connection capability to retry to connect instead of returning an error.

Set the INFORMIXCONTIME environment variable as follows:

setenv INFORMIXCONTIME *value*

value represents the minimum number of seconds spent in attempts to connect to each server.

For example, set INFORMIXCONTIME to 60 seconds as follows:

C shell:
```
setenv INFORMIXCONTIME 60
```

Bourne shell:
```
INFORMIXCONTIME=60
export INFORMIXCONTIME
```

If INFORMIXCONTIME is set to 60 and INFORMIXCONRETRY is set to 3, as shown in this appendix, attempts to connect to the server (after the initial attempt at 0 seconds) will be made at 20, 40, and 60 seconds, if necessary, before aborting. This 20-second interval is the result of INFORMIXCONTIME divided by INFORMIXCONRETRY.

If execution of the CONNECT statement involves searching DBPATH, the following rules apply:

- All appropriate servers in the DBPATH setting are accessed at least once, even though the INFORMIXCONTIME value might be exceeded. Thus, the CONNECT statement might take longer than the INFORMIXCONTIME time limit to return an error indicating connection failure or that the database was not found.
- The INFORMIXCONRETRY value specifies the number of additional connections that should be attempted for each server entry in DBPATH.

- The INFORMIXCONTIME value is initially divided among the number of server entries specified in DBPATH. Thus, if DBPATH contains numerous servers, you should increase the INFORMIXCONTIME value accordingly to allow for multiple connection attempts.

The default value for INFORMIXCONTIME is 15 seconds after the initial connection attempt. The INFORMIXCONTIME setting takes precedence over the INFORMIXCONRETRY setting; retry efforts could end after the INFORMIXCONTIME value has been exceeded, but before the INFORMIXCONRETRY value has been reached.

INFORMIXDIR

The INFORMIXDIR environment variable specifies the directory that contains the subdirectories in which your product files are installed. INFORMIXDIR must be set. If you have multiple versions of the **OnLine** or **INFORMIX-SE** database server, set INFORMIXDIR to the appropriate directory name for the version that you want to access. For information about when to set the INFORMIXDIR environment variable, see the *UNIX Products Installation Guide*, Version 6.0.

setenv INFORMIXDIR *pathname*

pathname is the directory path where the product files are installed.

Set the INFORMIXDIR environment variable to the following recommended installation directory:

C shell:

```
setenv INFORMIXDIR /usr/informix
```

Bourne shell:

```
INFORMIXDIR=/usr/informix
export INFORMIXDIR
```

INFORMIXSERVER

The INFORMIXSERVER environment variable specifies the default database server to which an explicit or implicit connection is made by **4GL**. The database server can be either **OnLine** or **INFORMIX-SE** and can be either local or remote.

```
setenv INFORMIXSERVER dbservername ──────────────────────┤
```

dbservername is the name of the default database server.

The value of INFORMIXSERVER must correspond to a valid *dbservername* entry in the **$INFORMIXDIR/etc/sqlhosts** file on the machine running the application. It must be specified using lowercase characters and cannot exceed 18 characters for the **OnLine** database server and cannot exceed 10 characters for the INFORMIX-SE database server. For example, to specify the **coral** database server as the default for connection, enter the following command:

C shell:

```
setenv INFORMIXSERVER coral
```

Bourne shell:

```
INFORMIXSERVER=coral
export INFORMIXSERVER
```

INFORMIXSERVER specifies the database server to which an application connects if the CONNECT DEFAULT statement is executed. It also defines the database server to which an initial implicit connection is established if the first statement in an application is not a CONNECT statement.

Note: *INFORMIXSERVER must be set even if the application or* **4GL** *does not use implicit or explicit default connections.*

INFORMIXSHMBASE

The INFORMIXSHMBASE environment variable affects only client applications connected to an **OnLine** server using the IPC shared-memory (ipcshm) communication protocol.

You use INFORMIXSHMBASE to specify where shared-memory communication segments are attached to the client process so that client applications can avoid collisions with other memory segments used by the

application. If you do not set INFORMIXSHMBASE, the memory address of the communication segments defaults to an implementation-specific value such as 0x800000.

setenv INFORMIXSHMBASE *value*

value is used to calculate the memory address.

OnLine calculates the memory address where segments are attached by multiplying the value of INFORMIXSHMBASE by 1024. For example, to set the memory address to the value 0x800000, set the INFORMIXSHMBASE environment variable as follows:

C shell:

```
setenv INFORMIXSHMBASE 8192
```

Bourne shell:

```
INFORMIXSHMBASE=8192
export INFORMIXSHMBASE
```

Resetting INFORMIXSHMBASE requires a thorough understanding of the application's use of memory. Normally you do not reset INFORMIXSHMBASE. For more information, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide,* Version 6.0.

INFORMIXSTACKSIZE

The INFORMIXSTACKSIZE environment variable affects only client applications connected to an **OnLine** server.

The **OnLine** administrator can set INFORMIXSTACKSIZE to specify the stack size (in kilobytes) that OnLine will use for a particular client session. Use INFORMIXSTACKSIZE to override the value of the **onconfig** configuration parameter STACKSIZE for a particular application or user.

setenv INFORMIXSTACKSIZE *value*

value is the stack size for SQL client threads in kilobytes.

For example, to decrease the INFORMIXSTACKSIZE to 20 kilobytes, enter the following command:

C shell:

```
setenv INFORMIXSTACKSIZE 20
```

Bourne shell:

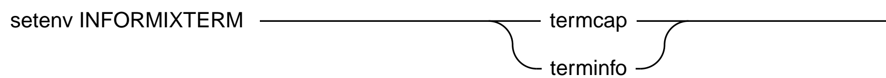
```
INFORMIXSTACKSIZE=20
export INFORMIXSTACKSIZE
```

Note: *If INFORMIXSTACKSIZE is not set, the stack size is taken from the* **OnLine** *configuration parameter STACKSIZE, or it defaults to an implementation-specific value. The default stack size value for the primary thread for an SQL client is 32 kilobytes for nonrecursive database activity.*

***Warning:** See the* INFORMIX-OnLine Dynamic Server Administrator's Guide, *Version 6.0, for specific instructions for setting this value. If you incorrectly set the value of INFORMIXSTACKSIZE, it can cause* **OnLine** *to crash.*

INFORMIXTERM

The INFORMIXTERM environment variable specifies whether **4GL** should use the information in the **termcap** file or the **terminfo** directory. INFORMIXTERM determines terminal-dependent keyboard and screen capabilities such as the operation of function keys, color and intensity attributes in screen displays, and the definition of window border and graphics characters.



If INFORMIXTERM is not set, the default setting is **termcap**. When **4GL** is installed on your system, a **termcap** file is placed in the **etc** subdirectory of **$INFORMIXDIR**. This file is a superset of an operating system **termcap** file.

You can use the **termcap** file supplied by Informix, the system **termcap** file, or a **termcap** file that you created yourself. You must set the TERMCAP environment variable if you do not use the default **termcap** file.

The **terminfo** directory contains a file for each terminal name that has been defined. It is supported only on machines that provide full support for the UNIX System V **terminfo** library. For details, see the Version 6.0 on-line machine notes for your machine.

Set the INFORMIXTERM environment variable to **terminfo** as follows:

C shell:
```
setenv INFORMIXTERM terminfo
```

Bourne shell:
```
INFORMIXTERM=terminfo
export INFORMIXTERM
```

Set the INFORMIXTERM environment variable to **termcap** as follows:

C shell:
```
setenv INFORMIXTERM termcap
```

Bourne shell:
```
INFORMIXTERM=termcap
export INFORMIXTERM
```

Note:** If INFORMIXTERM is set to* ***termcap*, you must set the UNIX environment variable TERMCAP; if INFORMIXTERM is set to* ***terminfo****, you must set the UNIX environment variable TERMINFO.*

ONCONFIG

The **OnLine** administrator can set the ONCONFIG environment variable (previously known as TBCONFIG), which contains the name of the UNIX file that holds the configuration parameters for OnLine. This file is read as input to either the disk-space or shared-memory initialization procedure.

```
setenv ONCONFIG filename
```

filename is the name of the file in **$INFORMIXDIR/etc** that contains the **OnLine** configuration parameters.

If you are not the administrator of an **OnLine** server, you need to set ONCONFIG only if more than one OnLine system is initialized in your **$INFORMIXDIR** directory, and you want to maintain multiple configuration files with different values. If you do not set ONCONFIG, the default is **onconfig**.

Each **OnLine** server has its own **onconfig** file that must be stored in the **$INFORMIXDIR/etc** directory. You might prefer to name **onconfig** so it easily can be related to a specific OnLine database server. For example, when the desired filename is **onconfig3**, you can set the ONCONFIG environment variable as follows:

C shell:
```
setenv ONCONFIG onconfig3
```

Bourne shell:
```
ONCONFIG=onconfig3
export ONCONFIG
```

For more information, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide,* Version 6.0.

PSORT_DBTEMP

The PSORT_DBTEMP environment variable affects only client applications connected to an **OnLine** server.

PSORT_DBTEMP specifies a directory or directories where the **OnLine** server writes the temporary files it uses when performing a sort. See the DBSPACETEMP environment variable on page D-28 for more information on other places **OnLine** can write information during a sort.

This variable is used even if the environment variable PSORT_NPROCS is not set.

pathname is the name of the UNIX directory used for intermediate writes during a sort.

Set the PSORT_DBTEMP environment variable as follows to specify the directory, for example, **/usr/leif/tempsort**:

C shell:
```
setenv PSORT_DBTEMP /usr/leif/tempsort
```

Bourne shell:
```
PSORT_DBTEMP=/usr/leif/tempsort
export PSORT_DBTEMP
```

For maximum performance, specify directories that reside in file systems on different disks.

You also might want to consider setting the environment variable DBSPACETEMP to place temporary files used in sorting in dbspaces rather than operating system files. See the discussion of the DBSPACETEMP environment variable on page D-28.

PSORT_NPROCS

The PSORT_NPROCS environment variable affects only client applications connected to an **OnLine** server.

PSORT_NPROCS enables the Psort parallel-process sorting package to improve performance on **OnLine**. The setting defines the upper limit for the number of threads used to sort a query.

value specifies the maximum number of threads to be used to sort a query.

Set the PSORT_NPROCS environment variable as follows to specify the maximum value:

C shell:
```
setenv PSORT_NPROCS 4
```

Bourne shell:
```
PSORT_NPROCS=4
export PSORT_NPROCS
```

To maximize the effectiveness of Psort, set PSORT_NPROCS to the number of available processors in the hardware. If PSORT_NPROCS is set to zero, Psort uses three as the default number of threads.

Use the following command to disable Psort:

C shell: `unsetenv PSORT_NPROCS`

Bourne shell: `unset PSORT_NPROCS`

For additional information about the PSORT_NPROCS environment variable, see *INFORMIX-OnLine Dynamic Server Administrator's Guide,* Version 6.0.

SQLEXEC

The SQLEXEC environment variable specifies the location of the Version 6.0 relay module executable that allows a Version 4.1 or earlier client to communicate indirectly with a local Version 6.0 **OnLine** or **INFORMIX-SE** database server. You must, therefore, set SQLEXEC only if you want to establish communications between a Version 4.1 or earlier client and a Version 6.0 database server.

setenv SQLEXEC *pathname*

pathname specifies the pathname for the relay module.

Set SQLEXEC as follows to specify the full pathname of the relay module, which is in the **lib** subdirectory of your **$INFORMIXDIR** directory:

C shell: `setenv SQLEXEC $INFORMIXDIR/lib/sqlrm`

Bourne shell:
```
SQLEXEC=$INFORMIXDIR/lib/sqlrm
export SQLEXEC
```

If you set the SQLEXEC environment variable on the C shell command line instead of in your **.login** or **.cshrc** file, you must include curly braces around the existing INFORMIXDIR, as follows:

C shell: `setenv SQLEXEC ${INFORMIXDIR}/lib/sqlrm`

For information on the relay module, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide,* Version 6.0, or the *INFORMIX-SE Administrator's Guide,* Version 6.0.

SQLRM

In Version 6.0, if the system administrator is configuring a client/server environment in which a Version 4.1 **4GL** client accesses a local Version 6.0 database server, the SQLRM environment variable must be *unset* before SQLEXEC can be used to spawn aVersion 6.0 relay module.

Unset SQLRM as follows:

C shell: `unsetenv SQLRM`

Bourne shell: `unset SQLRM`

For information on the relay module, see the *INFORMIX-OnLine Dynamic Server Administrator's Guide,* Version 6.0, or the *INFORMIX-SE Administrator's Guide*, Version 6.0.

SQLRMDIR

In Version 6.0, if the database administrator is configuring a client/server environment in which a Version 4.1 **4GL** client accesses a local Version 6.0 database server, the SQLRM environment variable must be *unset*.

Unset SQLRMDIR as follows:

C shell: `unsetenv SQLRMDIR`

Bourne shell: `unset SQLRMDIR`

NLS Environment Variables

The following variables apply to Native Language Support, and are documented in Appendix E, ''Native Language Support Within INFORMIX-4GL.''

NLS Environment Variable	Page
COLLCHAR	E-18
DBAPICODE	E-23
DBNLS	E-16
LANG	E-25
LC_COLLATE	E-27
LC_CTYPE	E-29
LC_MONETARY	E-31
LC_NUMERIC	E-35

UNIX Environment Variables

Informix products also rely on the correct setting of certain standard UNIX system environment variables. The PATH and TERM environment variables must always be set. You also might have to set the TERMCAP or TERMINFO environment variable to use **4GL** effectively.

PATH

The PATH environment variable tells the shell the order in which to search directories for executable programs. You must include the directory that contains **4GL** in your PATH environment variable before you can use **4GL**.

```
                              :
setenv PATH ──────────────┬─ pathname ─┬────────────┤
                          └─────────────┘
```

pathname specifies the search path for executables.

You can specify the correct search path in various ways. Be sure to include a colon between the directory names.

The following example uses the explicit path **/usr/informix**. This path must correspond to the INFORMIXDIR setting. The C shell example is valid for **.login** or **.cshrc** files.

C shell:
```
setenv PATH $PATH:/usr/informix/bin
```

Bourne shell:
```
PATH=$PATH:/usr/informix/bin
export PATH
```

The following example specifies **$INFORMIXDIR** instead of **/usr/informix**. It tells the shell to search the directories that were specified when INFORMIXDIR was set. The C shell example is valid for **.login** or **.cshrc** files.

C shell:
```
setenv PATH $PATH:$INFORMIXDIR/bin
```

Bourne shell:
```
PATH=$PATH:$INFORMIXDIR/bin
export PATH
```

You might prefer to use this version to ensure that your PATH entry does not contradict the path that was set in INFORMIXDIR, and so that you do not have to reset PATH whenever you change INFORMIXDIR.

Note: *If you set the PATH environment variable on the C shell command line instead of in your **.login** or **.cshrc** file, you must include curly braces with the existing INFORMIXDIR and PATH, as follows:*

C shell:
```
setenv PATH ${INFORMIXDIR}/bin:${PATH }
```

TERM

The TERM environment variable is used for terminal handling. It enables **4GL** to recognize and communicate with the terminal you are using.

```
setenv TERM type
```

type specifies the terminal type.

The terminal type specified in the TERM setting must correspond to an entry in the **termcap** file or **terminfo** directory. Before you can set the TERM environment variable, you must obtain the code for your terminal from the **OnLine** or **INFORMIX-SE** administrator.

For example, to specify the vt100 terminal, set the TERM environment variable as follows:

C shell:

```
setenv TERM vt100
```

Bourne shell:

```
TERM=vt100
export TERM
```

TERMCAP

The TERMCAP environment variable is used for terminal handling. It tells **4GL** to communicate with the **termcap** file instead of the **terminfo** directory.

```
setenv TERMCAP pathname
```

pathname specifies the location of the **termcap** file.

The **termcap** file contains a list of various types of terminals and their characteristics. Set TERMCAP as follows:

C shell:

```
setenv TERMCAP /usr/informix/etc/termcap
```

Bourne shell:

```
TERMCAP=/usr/informix/etc/termcap
export TERMCAP
```

Note: *If you set the TERMCAP environment variable, be sure that the INFORMIXTERM environment variable is set to the default,* ***termcap****.*

TERMINFO

The TERMINFO environment variable is used for terminal handling. It is supported only on machines that provide full support for the UNIX System V **terminfo** library.

setenv TERMINFO /usr/lib/terminfo

TERMINFO tells **4GL** to communicate with the **terminfo** directory instead of the **termcap** file. The **terminfo** directory has subdirectories that contain files pertaining to terminals and their characteristics.

Set TERMINFO as follows:

C shell:
```
setenv TERMINFO /usr/lib/terminfo
```

Bourne shell:
```
TERMINFO=/usr/lib/terminfo
export TERMINFO
```

Note: *If you set the TERMINFO environment variable, you also must set the INFORMIXTERM environment variable to* ***terminfo****.*

Appendix

E

Native Language Support Within INFORMIX-4GL

This appendix discusses the Native Language Support (NLS) features that are included in the 6.0 release of 4GL.

This appendix is organized as follows:

- Overview of NLS
- The Non-NLS Environment
- NLS Environments
- NLS Features Supported in **4GL**
- Classification and Precedence of Environment Variables
- Database Storage of Environment Variables
- Meta-Environment Variables
- X/Open-Defined Variables
- Informix-Defined Language and Formatting Variables
- LOAD and UNLOAD Statements
- FORMAT and USING
- Multiple Locale Support
- Language Supplements

Overview of NLS

Native Language Support is based on the X/Open Portability Guide Version 3 (XPG3) standard, which specifies a means for localization of software to European geographical regions.

A useful NLS concept is that of the *locale*; a locale specifies, by way of NLS environment variables, the language and formatting environment of an 4GL user (the *user locale*) or of a database at the time of its creation (the *database locale*). The LANG environment variable establishes an overall locale such as German or French.

Each specific feature of a locale, pertaining to one aspect of the language environment, is referred to as *category*. NLS categories are specified with environment variables whose names start with the letters LC followed by the underbar (_) symbol. NLS categories supported in 4GL include:

- Sorting sequence of characters, also known as *collation* (LC_COLLATE)
- Character set (LC_CTYPE)
- Monetary formatting (LC_MONETARY)
- Numeric formatting (LC_NUMERIC)

LC_ variables do not have to be individually specified. The LANG variable establishes defaults for all LC_ variables. These defaults are appropriate for most users in the geographic region implied by LANG. One or more LC_ variables are used to fine-tune particular features of the locale.

Note: *Since the LANG variable specifies values for LC_COLLATE, LC_CTYPE, LC_MONETARY, and LC_NUMERIC, references in the text to setting any of the LC_ variables can either mean setting the LC_ variable directly or setting it indirectly by way of LANG.*

Informix-Defined Environment Variables

In addition to standard NLS categories, Informix provides its own environment variables with the 6.0 tools.

Supported Informix-created environment variables control:

- On/off status of the NLS feature set, and level of required database-user locale matching (DBNLS).
- On/off status of implicit collation-sequence mapping between user and server (COLLCHAR).
- Character set translation between the database and non-ASCII terminal equipment (DBAPICODE).

- Location of files that specify error messages, menus, and month and day names that are translated into a national language (DBLANG and DBFORM).
- Numeric, monetary, and date formatting (DBFORMAT, DBMONEY and DBDATE). The Informix-created variables predate the XPG3 NLS variables, and so are supported for reasons of backwards compatibility.

NLS Character Sets

The NLS character sets are provided to meet the needs of European countries. Most European languages contain various characters that are not found in English, such as eszet (ß), a-umlaut (ä), and enye (ñ).

The ASCII character set used for English is a standard representation system for characters on computers. Any ASCII character can be represented with seven bits of data, of which there are 128 possible combinations. Each of these combinations has a numeric value between 0 and 127. For example, the capital letter B has the ASCII value 66.

It would seem that 128 possible ASCII values would provide sufficient room for representing non-English characters. However, the ASCII set has to include all CONTROL characters, punctuation, numeric digits, uppercase and lowercase alphabetic letters, and arithmetic symbols. There are no unused ASCII values for assignment to special European characters. ASCII characters and their ASCII values are listed in Appendix G, "The ASCII Character Set."

With the elimination of the need for computer systems to include a *parity* bit in 8-bit computer bytes, an eighth bit has become available for representing characters. This doubles the number of available character representation values, from 128 to 256. Particular computer manufacturers have created their own 8-bit character sets which work with their own computer equipment but not others. The first 128 values always correspond to ASCII values, but the remaining 128 values are assigned arbitrarily.

The International Standards Organization (ISO) has created two standard 8-bit character sets to support European alphabets uniformly across computer platforms. These are the ISO 8859-1 and ISO 8859-2 character sets, which support Western European and Eastern European languages, respectively.

NLS character set support provides, by way of the LC_CTYPE and DBAPICODE environment variables, the ability to:

- Specify a character set for representing the character data in a database.
- Create user-defined names such as table and column names from the set of characters supported by the database.

- Specify a different character set for the computer monitor or printer, and provide the means to translate characters between the database and the monitor or printer.
- Utilize built-in capitalization rules that assure correct conversion between uppercase and lowercase characters within a character set.

Note: *There are several possible standards for Asian and Arabic language representation, which are not supported by way of NLS. Asian and Arabic language support will require 16-bit character (or larger) representation. In the future these languages will be supported by way of a different standard called GLS (Global Language Support), which is an extension of the NLS standard.*

With the introduction of non-English characters into data, there arises a need for specifying how character data is to be alphabetized. 4GL 6.0 introduces the capability to sort and compare character data in a collating sequence specified by the LC_COLLATE category in the locale.

To accomplish this, two new data types are utilized by the server in databases created for NLS environments: NCHAR and NVARCHAR. These data types are not directly definable by the 4GL programs. Rather, using a process called *implicit mapping*, the server substitutes the data type NCHAR when the program defines or accesses a CHAR column, and substitutes the data type NVARCHAR when the program defines or accesses a VARCHAR column.

The effect of implicit mapping is to allow you to define the same character data types as you have in the past (pre-Version 6.0) in your applications, but have the system automatically treat this character data as sorted according to the locale.

The server's data types NCHAR and NVARCHAR are identical to CHAR and VARCHAR respectively, except that data in an NCHAR or NVARCHAR column is sorted and compared according to the LC_COLLATE setting in the locale, rather than according to the default collation (US English ASCII order).

The Non-NLS Environment

Depending on the settings in the environment variables DBNLS and COLLCHAR, the user operates in either the *Non-NLS environment* or one of three NLS environments. The three NLS environments are *Implicit NLS*, *Explicit NLS*, and *Open NLS*. These are discussed in the next section.

The Non-NLS environment is specified by unsetting the DBNLS variable (or equivalently, setting it to zero). The Non-NLS environment is equivalent to using a pre-6.0 version of 4GL. With the Non-NLS environment active, the following are true:

- The NLS environment variables LC_COLLATE, LC_CTYPE, LC_MONETARY, and LC_NUMERIC have no effect.
- All character data is sorted and compared according to US English collation.
- Only characters from the ASCII character set may be used in identifiers (such as database and table names).
- The default numeric and monetary formats (in the absence of DBFORMAT and DBMONEY settings) are ANSI compliant.

A database that is created when one of the NLS environments is active, known as an *NLS database*, cannot be accessed while in the Non-NLS environment.

NLS Environments

The three NLS environments (Implicit, Explicit and Open) are specified by combinations of the DBNLS and COLLCHAR environment variables in which DBNLS is set to a value of 1 or 2.

With any of the three NLS environments active, the following are true:

- The NLS environment variables LC_COLLATE, LC_CTYPE, LC_MONETARY, and LC_NUMERIC are considered in various operations.
- Character data columns are sorted and compared according to the collation sequence specified by the LC_COLLATE variable.
- Characters from the character set specified by LC_CTYPE are available for use in identifiers, in addition to the ASCII character set.
- Monetary and numeric formats specified by LC_MONETARY and LC_NUMERIC become active in display, input, printing, loading, and unloading of values, provided that DBFORMAT and DBMONEY are not set.

If any of the three NLS environments is active when a database is created, the database will be permanently associated with the collation sequence (LC_COLLATE) and character set (LC_CTYPE) that are current at the time of creation, and permanently designated as an NLS database.

A database that is created when the Non-NLS environment is active, known as a *non-NLS Database*, can be accessed while in any of the NLS environments. However, none of the NLS variables take effect when working in a non-NLS database. To avoid confusion, it is recommended that you unset DBNLS before working with a non-NLS database.

There are two distinctions between the three NLS environments:

1. Whether or not the program is required to access an NLS database with the same collation sequence (LC_COLLATE) and character set (LC_CTYPE) variable values specified in the current environment as were specified at the time of database creation.
2. Whether or not the user can define character columns that are sorted and compared in US English in a non-English NLS database.

All version 6.0 Informix tools can utilize the Implicit NLS and Open NLS environments. All version 6.0 Informix tools except 4GL can utilize the Explicit NLS environment. Implicit NLS is the recommended NLS environment, as it provides NLS capabilities without disruption to existing applications or risk of data corruption.

Implicit NLS

Implicit NLS is the environment defined by the DBNLS environment variable set to 1 and the COLLCHAR variable also set to 1. This is the standard NLS environment in which 4GL operates. All character columns defined by a program in the Implicit environment appear to the program as type CHAR (or VARCHAR, if variable-length), but are interpreted by the server as type NCHAR (or NVARCHAR) if the database is non-US-English. This is referred to as *implicit mapping*. Thus, all database columns sort according to the LC_COLLATE setting in the locale. Such columns are referred to as *locale-sorted*. Columns that are not locale-sorted cannot be created in the Implicit NLS environment.

The advantage of the Implicit environment for a programmer is that existing references to CHAR and VARCHAR data types do not have to be changed when an application is moved from a non-NLS to an NLS environment. Locale-sorting of character data becomes available without modification. Also, the programmer does not need to be concerned with which columns are locale-sorted and which columns are not.

The Implicit NLS environment, like Explicit NLS, mandates *consistency checking* between the current LC_COLLATE and LC_CTYPE values and those that were saved with the database at the time of database creation. Access is

not permitted to an NLS database with different LC_COLLATE and LC_CTYPE settings from the current environment's settings. This helps prevent certain kinds of data corruption.

Explicit NLS

Explicit NLS is the environment defined by the DBNLS environment variable set to 1 and the COLLCHAR variable unset. This NLS environment is unavailable to 4GL. In the Explicit environment, character columns can be defined by the user that are not implicitly mapped, that is, always sort as CHAR (or VARCHAR) data, regardless of locale. Columns that are to be locale-sorted are defined by the user explicitly as NCHAR (or NVARCHAR). Columns that are to be US English sorted are defined by the user as CHAR or VARCHAR, and these are not mapped to NCHAR and NVARCHAR at the server.

The Explicit NLS environment, like Implicit NLS, mandates consistency checking between the current LC_COLLATE and LC_CTYPE values and those that were saved with the database at the time of database creation. Access is not permitted to an NLS database with different LC_COLLATE and LC_CTYPE settings from the current environment's settings.

Open NLS

Open NLS is the environment defined by the DBNLS variable set to 2 and the COLLCHAR variable set to 1. Open NLS differs from Implicit and Explicit NLS in that the Open environment the system does not perform consistency checking when access to an NLS database is attempted. Any combination of settings of LC_COLLATE and LC_CTYPE is permitted when accessing an NLS database in the Open environment.

Consistency checking is normally a desirable feature, as it prevents data corruption such as would occur if French-sorted data were appended to a German-sorted character column or characters peculiar to Spanish were allowed in the names of tables in an Italian database. However, there are three situations in which overriding the consistency checking feature by means of specifying the Open environment are desirable:

1. When users of non-Informix tools (or Informix tools prior to version 6.0) need to access data in an NLS database stored in an Informix database engine, they would specify the Open environment.
2. When data is unloaded from an NLS database in one locale, and loaded to an NLS database in another locale.

3. When data is unloaded from a non-NLS database and loaded to an NLS database or vice versa.

Open NLS is similar to Implicit NLS in that implicit mapping is performed. All character columns created are locale-sorted, that is, NCHAR or NVARCHAR.

Summary of Environments

In the following table, behavior of NLS databases in Open NLS, Implicit NLS, Explicit NLS and Non-NLS environments is contrasted with the behavior of non-NLS databases in those environments:

Environment	Settings	Property	NLS database	Non-NLS database
Open NLS	DBNLS=2	program can access	Yes	Yes
	COLLCHAR=1	program can create	Yes	No
Implicit NLS	DBNLS=1	program can access	Yes	Yes
	COLLCHAR=1	program can create	Yes	No
Explicit NLS	DBNLS=1	program can access	Yes	Yes
	COLLCHAR unset	program can create	Yes	No
Non NLS	DBNLS=0 or	program can access	No	Yes
	DBNLS unset	program can create	No	Yes

Figure E-1 Database Access In Open, Implicit, and Explicit Environments

NLS Features Supported in INFORMIX 4GL 6.0

NLS features supported in 4GL version 6.0 include:

- Character data sorting and comparison according to the rules of a national language locale.
- Use of extended-ASCII characters permitted in user-defined names such as database, table, and column names.
- Nationalized money and numeric decimal formats in reports, screen forms, and data assignment statements.
- Character conversion between database data and national language specific keyboards and screens.
- The ability for different users to simultaneously access, on the same server, databases with different locale settings.

Figure E-2 and Figure E-3 present an overview of the affected data types and 4GL statements and keywords.

Data Type	Impact
CHAR	Transparently maps to NCHAR if 4GL is running in Implicit mode against an NLS database.
VARCHAR	Transparently maps to NVARCHAR if 4GL is running in Implicit mode against an NLS database.
NCHAR	Unavailable data type in 4GL.
NVARCHAR	Unavailable data type in 4GL.
DECIMAL	Display depends on values in DBFORMAT, DBMONEY, and LC_NUMERIC (highest to lowest precedence).
SMALLFLOAT	Display depends on values in DBFORMAT, DBMONEY, and LC_NUMERIC (highest to lowest precedence).
FLOAT	Display depends on values in DBFORMAT, DBMONEY, and LC_NUMERIC (highest to lowest precedence).
MONEY	Display depends on values in DBFORMAT, DBMONEY, and LC_MONETARY (highest to lowest precedence).
DATE	Separator symbol and order of month, day, and year depends on the value in DBDATE. Display of language-specific month and day names depends on installation of message files, whose location is referenced by DBLANG.
DATETIME	Display of language-specific month and day names depends on the installation of message files, whose location is referenced by DBLANG.

Figure E-2 Impact of NLS Support on Data Types

Statement or Keyword	IMPACT
LOAD	The LOAD statement expects incoming text files to be in the format specified by the NLS and Informix-defined environment variables.
UNLOAD	Text files produced by an UNLOAD are output in the format specified by NLS and Informix-defined environment variable values, but without thousands separators.
USING	Interpretation of format strings is dependent on settings in DBFORMAT, DBMONEY, DBDATE, LC_NUMERIC, and LC_MONETARY.
CREATE TABLE, ALTER TABLE	CHAR and VARCHAR columns defined in Implicit or Open NLS environments are created as NCHAR and NVARCHAR.
FORMAT	Same as USING, except that FORMAT does not support currency symbols.
ORDER BY, MATCHES, WHILE, INCLUDE, and IF	Comparisons of character values with NLS active are based on collation sequences defined by LC_COLLATE.
DISPLAY	Same as USING.
INPUT	Same as USING.
LET	Conversions between character and numeric, monetary or date values are dependent on settings in DBFORMAT, DBMONEY, DBDATE, LC_NUMERIC, and LC_MONETARY.
UPSHIFT and DOWNSHIFT	Translations between upper and lowercase are specified by LC_CTYPE.
DATE	The date displayed contains month and day names specified by the message files pointed to by DBLANG.
CALL (to C function)	Called C functions can include locale-specific characters in identifiers, if the C compiler can support these.

Figure E-3 Impact of NLS Support on Statements and Keywords

Classification and Precedence of Environment Variables

The environment variables used by Informix servers and tools, including 4GL, are either *X/Open-defined* variables or *Informix-defined* variables. X/Open-defined variables include LANG and variables that start with the characters LC_. Collectively the X-Open defined variables specify the *locale.* Informix-defined variables are not in the X/Open standard, and include COLLCHAR and variables that start with the characters DB.

The LANG and LC_ variables, along with the Informix-defined variables DBFORMAT, DBDATE, and DBMONEY, together specify the language and formatting environment of the user. These variables are referred to as language and formatting variables. They define the following aspects of the environment:

- Sort order of character data (LC_COLLATE)
- Valid character set for identifiers (LC_CTYPE)
- Monetary data format (DBFORMAT, DBMONEY and LC_MONETARY)
- Numeric data format (DBFORMAT, DBMONEY and LC_NUMERIC)
- Date format (DBDATE)

The DBNLS, COLLCHAR, DBAPICODE, and DBLANG environment variables are referred to as meta-environment variables. They specify the following aspects of the meta-environment:

- Whether or not NLS is activated (DBNLS).
- Whether Explicit, Implicit, or Open NLS is active (DBNLS and COLLCHAR).
- The name of a character translation file for mapping characters between the database and terminal equipment (DBAPICODE).
- The location of message and menu form files (DBLANG and DBFORM).

These classifications are summarized in the following table:

	X/Open defined (locale)	Informix defined
Meta-Environment		DBNLS COLLCHAR DBAPICODE DBLANG DBFORM
Language and Formatting	LANG LC_COLLATE LC_CTYPE LC_MONETARY LC_NUMERIC	DBFORMAT DBMONEY DBDATE

It is important to distinguish between Informix-defined and XPG3-defined (locale) variables. This is because the locale variables rely on facilities provided by the computer manufacturer, and may vary in meaning from platform to platform. For example, the German locale may be specified as LANG DE on one system and LANG de on another, and the two may imply different collation, character sets, or formatting rules. Informix-defined variables, in contrast, have uniform syntax and meaning across platforms.

The following Informix-defined variables can be used without activating NLS, but interact with NLS variables because they pertain to monetary, numeric, and date formatting. They are covered in detail in Appendix D, "Environment Variables."

- DBLANG
- DBFORMAT
- DBMONEY
- DBDATE

The overall precedence hierarchy for language and formatting variables is as follows:

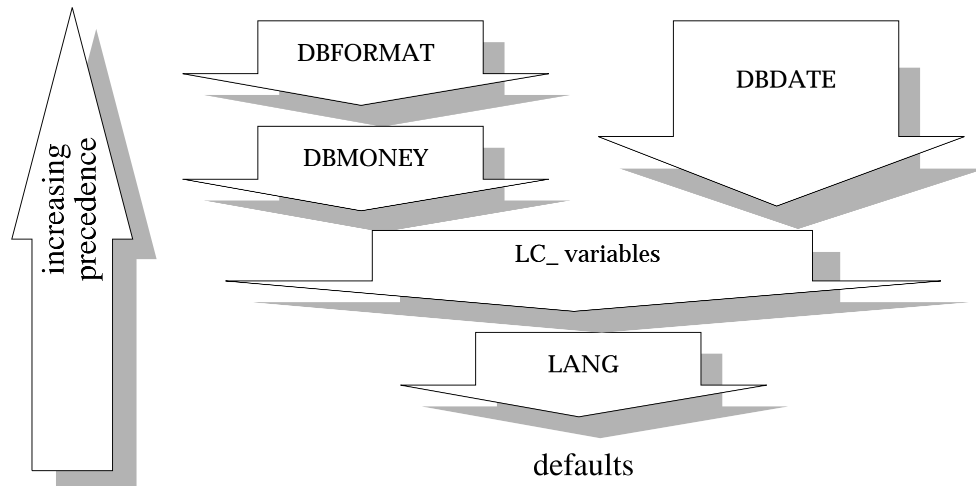


Figure E-4 Order of Precedence of Language and Formatting Environment Variables

The Informix-defined language and formatting variables—DBFORMAT, DBMONEY, and DBDATE—take the highest precedence, the LC_ variables intermediate precedence, and the LANG variable the lowest. Any of these will override internal defaults. Therefore, a DBFORMAT setting will override an LC_NUMERIC setting, which in turn will override LANG, which will override the default.

In Informix tools and engine products, the existing variables that specify language-specific or country-specific behavior need to be retained and given the highest priority, for the sake of existing user applications that use these variables.

In the XPG3 NLS standard, the LANG variable specifies a locale, and the LC_ variables are used to modify particular pieces of the locale that the user wants to be different from the standards for that locale. The LANG variable sets the default values of all of the LC_ variables. The user may override the LANG defaults for particular LC_ variables by setting those individually.

In the following example, the user is selecting the German language locale (LANG set to DE). However, the user then selects the German Swiss dictionary collation sequence to override the default German sequence (LC_COLLATE), and the ISO 8859/1 set of monetary formats for expressing money values in Swiss francs (LC_MONETARY) to override the default German monetary format:

```
setenv LANG DE
setenv LC_COLLATE DE_ch@dict
setenv LC_MONETARY CH.88591
```

The order in which environment variables is specified is not significant. However, the directory location where a locale variable is defined influences the precedence, as illustrated in the following diagram:

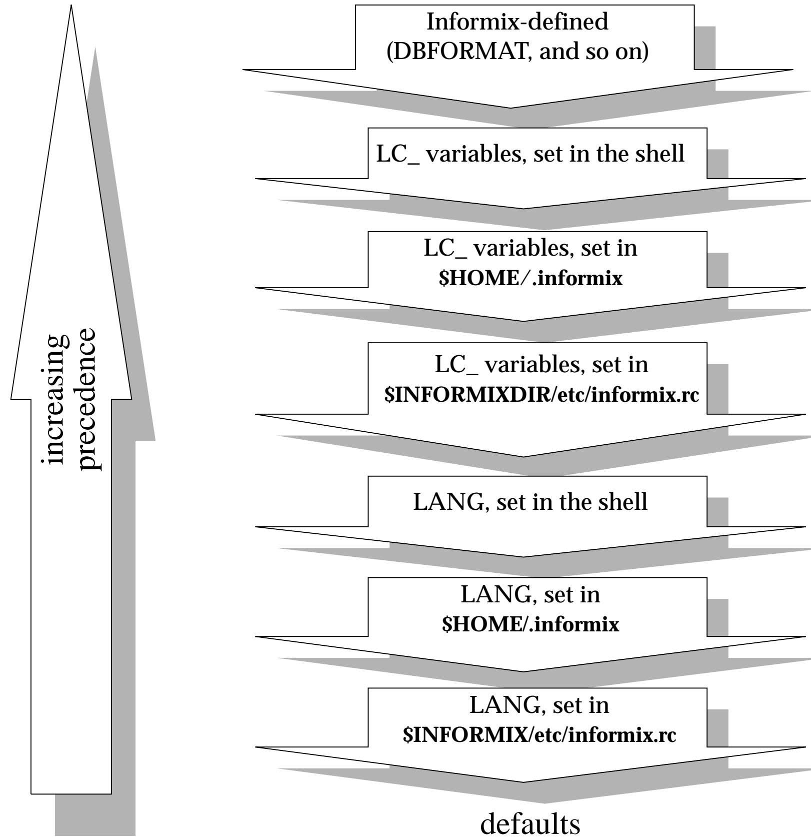


Figure E-5 Order of Precedence of Variables, Considering Where Set

1. The highest precedence goes to the value as defined in the environment (shell).
2. The second-highest precedence goes to the value as defined in the private environment-configuration file in the user's home directory (~/**.informix**). This is a private file where you can define all the environment variables that are used by Informix products. Using a configuration file reduces the number of environment variables that you must set at the command line or in a shell file.
3. The next-highest precedence goes to the value as defined in the common environment-configuration file (**$INFORMIXDIR/etc/informix.rc**). This is

the same as a private environment-configuration file in the user's home directory, only for shared use rather than private.

4. The lowest precedence goes to the default value.
5. The setting for any of the LC_ variables takes precedence over the setting for the LANG environment variable, no matter where they are set.

Database Storage of Environment Variables

When the database connection is established between a user session and the database server by way of the DATABASE statement in 4GL, the information in the local environment variables LC_COLLATE and LC_CTYPE will be transmitted with the request for database service. If a new database is being created, the database server saves these values in system tables inside the database. If the program is requesting access to an existing database, the server rejects the connection request if the current environment's LC_COLLATE and LC_CTYPE values do not match the saved database values (and the Implicit or Explicit, rather than Open, NLS environment is being employed). This process is referred to as *consistency checking.*

For 4GL, consistency checking takes place at compilation time for the LC_CTYPE variable and at run time for the LC_COLLATE variable.

These values are kept unchanged throughout the life of the database to ensure the consistent use of collating sequences and character sets. The LC_COLLATE and LC_CTYPE environment variable settings for a database cannot be changed; the data must be unloaded and reloaded into a different database to change collation or character set features of the database. Numeric and monetary format features of the locale can be changed as desired, because monetary and numeric data are saved in database tables in ANSI format. LC_MONETARY and LC_NUMERIC affect the display and input of data from the standpoint of the user, not internally at the server. However, they do affect the format of an ASCII file created by an UNLOAD operation, and the interpretation of the data in that file during a LOAD.

Meta-Environment Variables

Meta-environment variables are all Informix defined. They turn on and off features of the NLS environment and point to the locations of certain files that NLS uses.

DBNLS

The Informix-defined DBNLS environment variable enables or disables the NLS features implemented in the 6.0 tools and server products, and specifies the level of consistency checking to take place between user and database. The user activates the NLS capability in 4GL by setting DBNLS to 1 or 2. NLS capability is deactivated by unsetting DBNLS or setting it to 0, which is equivalent to unsetting.

DBNLS is set to 0, 1, or 2 as follows:

DBNLS is unset as follows:

Usage

If DBNLS is not set or set to 0, NLS functionality is not turned on, and the settings for the other NLS environment variables do not take effect. If DBNLS is set to 2 (and COLLCHAR set to 1, which is required), NLS functionality is activated but in the Open NLS environment. The Open NLS environment allows inconsistent locales between user and database.

Unlike LC_CTYPE and LC_COLLATE, the actual value of DBNLS is not saved with a database at creation time. However, the DBNLS setting at creation time permanently determines whether the database is an NLS database or a non-NLS database.

The meanings of the available settings for DBNLS are listed in the table below:

Value	Description
DBNLS = unset or DBNLS = 0	This is the default setting. It specifies the non-NLS environment. Only non-NLS databases can be created or accessed if the user environment has this setting. Any attempt to access an existing NLS database results in an error. Any databases created will not be NLS compatible. 4GL version 6.0 with this setting will behave exactly like non-NLS (pre-6.0) 4GL.
DBNLS = 1	This is the recommended setting for performing NLS work. This setting establishes either the Implicit (recommended) or Explicit NLS environment, depending on the setting of COLLCHAR. Either NLS or non-NLS databases can be accessed with DBNLS set to 1, but only NLS databases can be created. When accessing an existing NLS database, the user session LC_CTYPE and LC_COLLATE values must match the corresponding database values in order for access to be permitted. A database is locked into the current user environment values of the these two variables at database creation time.
DBNLS = 2	This setting specifies the Open NLS environment (in combination with a COLLCHAR setting of 1). Open NLS provides access to NLS databases regardless of whether or not the user session and database server settings of LC_COLLATE and LC_CTYPE match. The server obtains its settings of these variables from the database being accessed. New databases inherit the current locale of the user when they are created. Open NLS allows tools that are not aware of internal database locale settings to access NLS databases, and provides a means to load and unload data between dissimilar locales. Otherwise, this setting is not recommended. Character comparison results can be inconsistent between the user (for example, variable1 greater than variable2) and SQL queries issued to the server (ORDER BY, WHERE, MATCHES). Database data can be corrupted using Open NLS when the user updates the database, because the active environment settings can differ from the settings in which the database is intended to operate. For example, column names could be created with one valid character set active, and then become illegal when the active character set changes.
DBNLS = any value other than 0, 1, 2 or unset	This is an invalid setting. A run-time error will be returned.

Figure E-6 Values for the DBNLS environment variable

Operations on non-NLS databases while in an NLS environment follow the same rules as operations in pre-6.0 4GL. These include:

- Sorting order based on ASCII character code
- User defined names allowed to contain only ASCII characters

COLLCHAR

The Informix-defined COLLCHAR environment variable turns on or off the implicit mapping feature, which is necessary for using the Implicit NLS and Open NLS environments. COLLCHAR determines whether the data types NCHAR and NVARCHAR can be accessed directly by a client application (COLLCHAR = unset), or whether they are accessed by way of implicit mapping of CHAR and VARCHAR data types (COLLCHAR = 1). COLLCHAR has no effect in the non-NLS environment (DBNLS set to 0 or unset), in which the NCHAR and NVARCHAR data types are inaccessible.

Implicit NLS is the recommended NLS environment, as it provides NLS capabilities without disruption to existing applications or risk of data corruption. Refer to the discussion of Implicit, Explicit and Open NLS starting on page E-6.

4GL version 6.0 does not support the Explicit NLS environment. COLLCHAR must be set to 1 when using 4GL with an NLS database.

The user activates implicit mapping in by setting COLLCHAR to 1. Note that 0 is not an acceptable value for COLLCHAR, and may cause unpredictable results.

COLLCHAR is set to 1 as follows:

```
setenv COLLCHAR 1
```

COLLCHAR is unset as follows:

```
unsetenv COLLCHAR
```

Note: *There is a significant performance penalty associated with using locale-sorted (NCHAR and NVARCHAR) data types at the server. When there is no need for sorting according to the locale-specified collating sequence, users should create non-NLS databases rather than NLS databases.*

Usage

The meanings of the available settings for COLLCHAR are listed in the table below:

Value	Description
COLLCHAR =unset	This is the default value. It specifies the non-NLS environment.
COLLCHAR = 1	This setting specifies Implicit NLS when DBNLS=1 and Open NLS when DBNLS=2. This setting activates implicit mapping, in which the database server maps all incoming CHAR requests to NCHAR, and outgoing requests to the client back to CHAR again. When DBNLS=1 (Implicit), this COLLCHAR setting causes consistency checking of LC_COLLATE and LC_CTYPE settings between the user environment and the database. When DBNLS=2 (Open), this COLLCHAR setting will enable access to NLS databases with different LC_COLLATE and LC_CTYPE settings than the user session. This COLLCHAR setting requires a DBNLS setting of 1 or 2 to have any effect.
COLLCHAR = any value other than 1 or unset	A run-time error will be returned

Figure E-7 ***Values for the COLLCHAR environment variable***

The relationship between DBNLS and COLLCHAR is the following:

	COLLCHAR = unset (implicit mapping off)	**COLLCHAR = 1 (implicit mapping on)**
DBNLS = **unset or** DBNLS = **0**	**Non-NLS environment**. Only non-NLS databases can be accessed and created by way of these settings.	Non-standard combination of settings, but the same effect as the Non-NLS environment (COLLCHAR=unset).
DBNLS = **1**	**Explicit NLS environment.** Invalid combination of settings for 4GL. A user can create both locale sorted and US English sorted columns in the same database. LC_CTYPE and LC_COLLATE of user must match values of database for access.	**Implicit NLS environment**. This is the recommended environment for NLS work. Server will map incoming CHAR requests to NCHAR, outgoing NCHAR to CHAR. LC_CTYPE and LC_COLLATE of user must match values of database for access.
DBNLS = **2**	Invalid combination of settings. For Open NLS, COLLCHAR must be set to 1, or results will be unpredictable.	**Open NLS environment**. LC_CTYPE and LC_COLLATE settings of the user are not considered, except when creating a database. The server will map incoming CHAR requests to NCHAR, and outgoing NCHAR to CHAR.

Figure E-8 ***DBNLS/COLLCHAR Cross Reference Table***

In the Implicit NLS environment, 4GL behaves as if all character columns were defined in the language-specific collation sequence. Although defined as CHAR or VARCHAR fields by the user, the behavior of all character columns in the Implicit environment is as if they were defined as NCHAR or NVARCHAR, that is, using locale-specified collation.

The details of this process are is illustrated with the following sample **4GL** program excerpt and description of the effect of the **4GL** instructions on an NLS database:

```
CREATE DATABASE gödel
CREATE TABLE t1 (c1 CHAR(20))
LOAD FROM "names.und" INSERT INTO t1
DECLARE c1 CURSOR FOR
    SELECT c1 FROM t1 ORDER BY c1
FETCH c1 INTO my_name
```

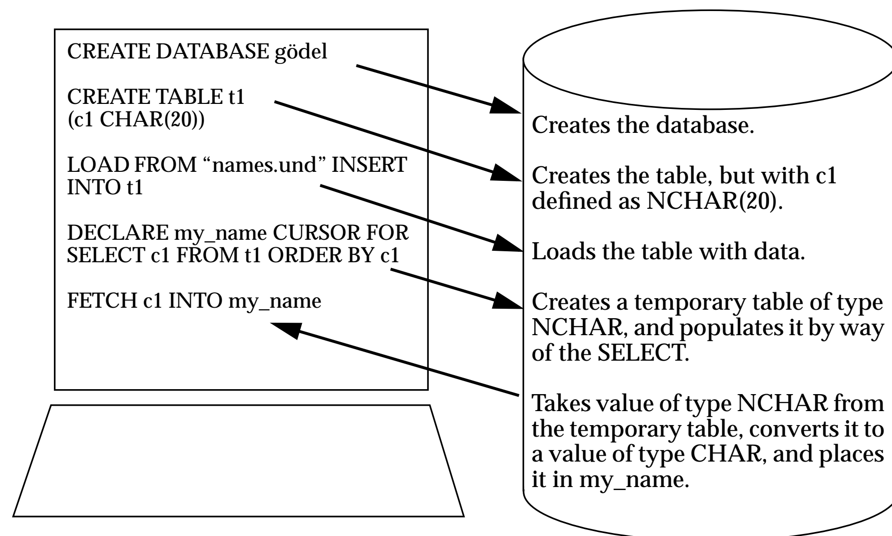


Figure E-9 ***llustration of Implicit Mapping Process Between 4GL and Server***

The difference between Implicit NLS (DBNLS = 1, COLLCHAR = unset) and a non-NLS environment (DBNLS=0 or unset) is illustrated with the example below, in which a database table and character column are created first in Implicit NLS and then in non-NLS.

Implicit NLS settings and the German locale (LANG=DE) are specified at the operating system prompt:

```
setenv DBNLS 1
setenv COLLCHAR 1
setenv LANG DE
isql
```

A new database is created with the following statement:

```
CREATE DATABASE test1
```

A new table (**länder**— German for *countries*) containing two columns, one named **land** (German for *country*) and one named **hauptstadt** (German for *capital*) are created by way of the following statement:

```
CREATE TABLE länder
(land CHAR(30), hauptstadt CHAR(28))
```

Since this is an NLS database (created with DBNLS set to 1) and implicit mapping is active (COLLCHAR is set to 1), the ***land*** and ***hauptstadt*** columns are created with data type NCHAR at the server. Any data saved in these columns will sort in German collating sequence.

To test this, we input the following data into the new table:

land (country) value	hauptstadt (capital) value
Österreich	Wien
Portugal	Lissabon
Luxemburg	Luxemburg

The following SQL command is executed:

```
SELECT * FROM länder
 ORDER BY land
```

The result of the query will be:

Luxemburg	Luxemburg
Österreich	Wien
Portugal	Lissabon

Note that Ö sorts between L and P, which is correct German alphabetization.

We return to the operating system prompt and enter the following setting, which specifies the non-NLS environment instead of Implicit:

```
unsetenv DBNLS
unsetenv COLLCHAR
```

We create a different database, this time called **test2**. We also create the **land** and **hauptstadt** columns again:

```
CREATE TABLE länder
(land CHAR(30), hauptstadt CHAR(28))
```

The same data values are input as before, and the SELECT query is executed. This time the query returns the following:

Österreich	Wien
Luxemburg	Luxemburg
Portugal	Lissabon

This is correct sorting for US ASCII in an NLS database.

***Note:** An NLS database established with DBNLS=1 or 2 (NLS active) and LANG=EN_us (US ASCII) does not collate the same as a non-NLS database (DBNLS=unset or 0). The collation within the 128-character ASCII character set will be the same in either case, but the 128 special characters will collate ahead of ASCII in the NLS database with the EN_us locale, and will collate behind ASCII in the non-NLS database.*

DBAPICODE

The Informix-defined DBAPICODE environment variable lets a computer peripheral whose character set differs from the database character set access the database. In this context, *peripheral* refers to a keyboard, monitor, or printer.

DBAPICODE specifies the character-mapping file between the peripheral and the database character set. In NLS databases, the database character set is defined in the LC_CTYPE environment variable. In non-NLS systems, the database character set is the default ASCII character set.

setenv DBAPICODE *mapfilename*

mapfilename names a character mapping file to be used for translation of characters in the database character set to the keyboard and monitor character set.

Usage

DBAPICODE specifies to the system the name of a mapping file created by the Informix utility **crtcmap**. With DBAPICODE set, 4GL communicates with the database server by using the ASCII character set, but interacts with the keyboard, monitor, or printer using the character set mapping file specified in DBAPICODE. If DBAPICODE is left unset, the system communicates with the keyboard and terminal by way of ASCII.

To use a specific DBAPICODE setting, there must be a mapping file for that character set in the message directory. For example, to use the character set **FR_fr.646**, there must be a mapping file called **mFR_fr.646**. This file should be located in the message directory **$INFORMIXDIR/msg**. If not, DBLANG or LANG must point to the message directory. Refer to the discussion of DBLANG on pages E-24 and D-18.

The **crtcmap** utility helps you create mapping files between database and peripheral character sets for use with NLS. It prepends **m** to the name of the mapping file it creates. For example, it renames the output file **FR_fr.646** as **mFR_fr.646**. The **crtcmap** utility is described in the *Informix Guide to SQL: Reference*, Version 6.0.

An example of setting DBAPICODE to a character set file is as shown:

```
setenv DBAPICODE mFR_fr.646
```

DBLANG

The DBLANG variable specifies the subdirectory of $**INFORMIXDIR** (or full pathname) in which the message files for the currently active language reside. Message files provide a set of error messages for the engine and tools that have been translated into a national language. Message files have the suffix **.iem**. DBLANG also points to the location of the character mapping file, as discussed under "DBAPICODE" on page E-23. Message files are obtained as part of language supplements, which include instructions specifying where the files should be installed and what DBLANG settings to specify.

For a complete discussion of the DBLANG variable, refer to Appendix D, "Environment Variables."

DBFORM

The DBFORM variable specifies the subdirectory of $**INFORMIXDIR** (or full pathname) in which the menu form files for the currently active language reside. Menu form files provide a set of language-translated menus to replace the standard 4GL menus. Menu form files have the suffix **.frm**. Menu form files are obtained as part of language supplements, which include instructions specifying where the files should be installed and what DBFORM settings to specify.

For a complete discussion of the DBFORM variable, refer to Appendix D, "Environment Variables."

X-Open Defined Variables

X-Open defined variables include LANG and variables that start with the characters LC_. These are also known as locale variables.

LANG

The value of the X/Open-defined LANG environment variable specifies the *locale* of the user's environment. A LANG setting specifies all of the following:

- A collating sequence.
- A set of translations between uppercase and lowercase.
- A legal character set for user-defined names.
- Formatting rules for monetary and numeric data.

The LANG setting implies standard values for the language environment that you can modify or override by setting LC_COLLATE, LC_CTYPE, LC_MONETARY, or LC_NUMERIC. The LC_ environment variables are described in the sections that follow. The setting for these LC_ categories always takes precedence over the LANG setting. For example, if a LANG setting for Swiss is specified, but LC_COLLATE is set for German, the Swiss settings apply to all categories in the locale except for character collation, which is German.

LANG should always be set to some locale value when creating or accessing NLS databases. The defaults for the LC_ variables when LANG is not set are unpredictable and vary between systems.

LANG identifies the required language, territory, and character set as illustrated in the syntax diagram below. The territory, character set and @modifier portions are optional and are not supported on some systems or with some LANG values.

The formal syntax for LANG is as follows:

```
setenv LANG language ──┬─────────────┬──┬──────────┬──┬────────────┬──┤
                        └─ _territory ┘  └─ .charset ┘  └─ @modifier ┘
```

language	a one to two character abbreviation such as *FR* for French, *DE* for German or *EN* for English.
_territory	specifies a dialect of a language, such as *DE_ch* (Swiss dialect of German) or *EN_uk* (British dialect of English).
.charset	specifies a character set other than ASCII, such as *.88591* (the ISO 8859/1 character set).
@modifier	specifies a collation sequence within a language and territory, such as @*dictionary* (dictionary order) or @*telephone* (telephone book order).

In practice, the syntax for LANG is likely to be one of the following:

```
setenv LANG language
```

or

```
setenv LANG language_territory
```

There is no standardization of LANG locale values between systems. Exact values to specify to obtain particular locale settings are particular to different computer systems, and also depend on which language supplements have been installed on your system.

Locale Files

Locale files installed on your system are what actually determine the set of correct values for the LANG and LC_ variables. The implementation varies between computer platforms but the purpose on all systems is the same. Locale files do the following:

- Translate a value that you set for a locale variable into a set of rules governing system behavior.
- Define the list of acceptable values that you can set each locale variable to.

For example, on the Sun platform there are locale files found in directories called **LC_COLLATE**, **LC_CTYPE**, **LC_MONETARY**, and **LC_NUMERIC**. These are locale directories. When you set LANG to DE on a Sun workstation, you are referencing a file named **DE** in each of these four locale directories. When you set the environment variable LC_COLLATE to DE@telephone, a file named **DE@telephone** is referenced in the **LC_COLLATE** directory.

Other computer manufacturers such as IBM and Hewlett-Packard use other schemes for translating a locale variable setting into a pointer to a set of rules or values. The set of acceptable values for each locale variable and the effects of particular settings are dependent on which computer system you are using. You can add to the set of valid LANG and LC_ variable settings on your system by installing language supplements.

The syntax diagrams for LANG and the LC_ variables in this manual suggest a range of possible settings for these variables, rather than an exact syntax for each variable that is applicable to all systems. Consult your language supplement documentation for specific settings.

LC_COLLATE

The X/Open-defined environment variable LC_COLLATE specifies a particular NLS collating sequence for the following:

- Sorting character data.
- Performing character string comparisons.
- Evaluating *regular expressions*, such as [A-Z], in MATCHES clauses.

LANG provides the default collation sequence if LC_COLLATE is not set. The value of LC_COLLATE, along with LC_CTYPE, is saved in an NLS database at database creation time, and consistency checked prior to database access. Refer to "Database Storage of Environment Variables" on page E-15.

If LC_COLLATE (directly or by way of LANG) is set to a non-US English value, this value changes the following features of the Informix database environment:

- Collating sequence
  - For an ORDER BY or GROUP BY clause in SELECT statements.
  - For index generation.
- Result set
  - Of expressions containing character fields in the WHERE clause of a SELECT statement.

The formal syntax for LC_COLLATE is as follows:

```
setenv LC_COLLATE language ──┬────────────┬─┬──────────┬─┬───────────┬──┤
                             └─_territory─┘ └─.charset─┘ └─@modifier─┘
```

language	a one to two character abbreviation such as *FR* for French, *DE* for German or *EN* for English.
_territory	specifies a dialect of a language, such as *DE_ch* (Swiss dialect of German) or *EN_uk* (British dialect of English).
.charset	specifies a character set other than ASCII, such as *.88591* (the ISO 8859/1 character set).
@modifier	specifies a collation sequence within a language and territory, such as *@dictionary* (dictionary order) or *@telephone* (telephone book order).

In practice, the syntax for LC_COLLATE is likely to be one of the following:

```
setenv LC_COLLATE language
```

or

```
setenv LC_COLLATE language_territory
```

or

```
setenv LC_COLLATE language_territory@modifier
```

There is no standardization of LC_ variable values between systems. Refer to the discussion of locale files on page E-26 for a better understanding of this syntax.

The following example sets the LC_COLLATE environment variable to specify the German telephone book sorting order:

```
setenv LC_COLLATE DE@telephone
```

An example of a result set influenced by the LC_COLLATE setting appears below. The following query will produce different result sets depending on whether the database is created with German or English collation:

```
SELECT hauptstadt FROM länder
WHERE land >= "Luxemburg" AND land <= "Portugal"
```

The query selects capitals of countries whose country names are alphabetically between Luxemburg and Portugal. With German collating, the capital of Austria (Vienna, or *Wien* in German) would be included in the result set, because the German word for Austria is *Österreich*. Österreich is between *Luxemburg* and *Portugal* in German; in US English in an NLS database it is not. Refer to the example in the section entitled "COLLCHAR" on page E-18.

LC_CTYPE

The X/Open-defined LC_CTYPE environment variable is used to specify which predefined set of characters can be legally contained in user-defined names. This includes the following:

- Names of databases, tables, and columns
- Names of views and synonyms
- Aliases
- Names of indexes and constraints
- Names of owners and users
- Cursor names and statement ID's
- Variable names
- Names of form specification files
- Names of report specification files

LC_CTYPE also defines what characters result from conversion of lowercase to uppercase letters, and vice versa.

Along with LC_COLLATE, the database server saves the LC_CTYPE setting at database creation time. The LC_CTYPE setting is used throughout the lifetime of the database; therefore a database saved with a particular LC_CTYPE value will always accept the same set of naming characters.

The formal syntax for LC_CTYPE is as follows:

```
setenv LC_CTYPE language [_territory] [.charset] [@modifier]
```

language	a one to two character abbreviation such as *FR* for French, *DE* for German or *EN* for English.
_territory	specifies a dialect of a language, such as *DE_ch* (Swiss dialect of German) or *EN_uk* (British dialect of English).
.charset	specifies a character set other than ASCII, such as *.88591* (the ISO 8859/1 character set).
@modifier	specifies a collation sequence within a language and territory, such as @*dictionary* (dictionary order) or @*telephone* (telephone book order).

In practice, the syntax for setting LC_CTYPE is likely to be one of the following:

```
setenv LC_CTYPE language
```

or

```
setenv LC_CTYPE language_territory
```

or

```
setenv LC_CTYPE language.charset
```

There is no standardization of LC_ variable values between systems. Refer to the discussion of locale files on page E-26 for a better understanding of this syntax.

For example, if the language environment for your system is defined as US English (LANG=EN_us), but you wish to allow German characters in user-defined names, you would issue the following command at the operating system prompt:

```
setenv LC_CTYPE DE
```

Subsequently, you could create a database whose name, **gödel**, contains the *ö* (o-umlaut) character and the statement would compile without error:

```
CREATE DATABASE gödel
```

Note: *Use of NLS character sets other than EN_us may require you to perform the UNIX command* `stty -istrip` *before accessing* **4GL** *on some systems. This command enables you to type in characters from the keyboard that are outside the ASCII character set.*

LC_MONETARY

The X/Open-defined LC_MONETARY environment variable is used to set the format of values of data type MONEY. This default format affects how monetary values are:

- Displayed and input on the screen
- Printed
- Input from ASCII files using LOAD
- Output to ASCII files using UNLOAD

LC_MONETARY specifies the locale-specific leading and trailing currency symbols, including their positions within a monetary value, and the decimal and thousands separators. Note that the decimal and thousands separators defined for monetary data by LC_MONETARY are distinct from the decimal and thousands separators defined for numeric data by LC_NUMERIC.

For example, the value 120.50 expressed as money appears as $120.50 if the locale is US English. However, by setting LC_MONETARY (or LANG) to UK English (EN_uk), the same value displays as £120.50, and by setting LC_MONETARY to German, the value displays as 120,50DM.

The setting in LC_MONETARY affects the following 4GL keywords:

- USING expression
- FORMAT attribute
- PRINT statement
- LET statement, where a character string is receiving a monetary value
- DISPLAY statement

LC_MONETARY utilizes locale files the way LANG and the other LC_ variables do. It does not directly specify currency and separator symbols the way DBMONEY and DBFORMAT do.

When LC_MONETARY is set and not overridden by DBFORMAT or DBMONEY, logic is employed by 4GL to determine issues such as the following:

- Whether a leading or trailing currency symbol is appropriate for this locale
- Whether the decimal portion of a monetary amount should be omitted, as for the Italian lira

The formal syntax for LC_MONETARY is as follows:

```
setenv LC_MONETARY language ──┬─────────────┬─┬──────────┬─┬───────────┬─┤
                              └─ _territory ─┘ └─ .charset ─┘ └─ @modifier ─┘
```

language	a one to two character abbreviation such as *FR* for French, *DE* for German or *EN* for English.
_territory	specifies a dialect of a language, such as *DE_ch* (Swiss dialect of German) or *EN_uk* (British dialect of English).
.charset	specifies a character set other than ASCII, such as *.88591* (the ISO 8859/1 character set).
@modifier	specifies a collation sequence within a language and territory, such as *@dictionary* (dictionary order) or *@telephone* (telephone book order).

In practice, the syntax for setting LC_MONETARY is likely to be one of the following:

```
setenv LC_MONETARY language
```

or

```
setenv LC_MONETARY language_territory
```

There is no standardization of LC_ variable values between systems. Refer to the discussion of locale files on page E-26 for a better understanding of this syntax.

Usage

The setting in LC_MONETARY determines the leading or trailing currency symbol, and the numeric and decimal separators. It adds the currency symbol and changes the separators displayed on the screen in a monetary field, and in the default format of a PRINT statement. For example, in a German locale the value 1234.56 prints as:

```
1234,56DM
```

DM stands for deutche marks. In a screen form with the French or German locale values active, input by the user will be expected to contain commas, not periods, as decimal separators.

The setting in LC_MONETARY also affects the way format strings in the FORMAT attribute and the USING clause are interpreted. In these format strings, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma symbol (,) is a placeholder for the thousands separator specified by environment variables. The dollar sign ($) is a placeholder for the leading currency symbol. The at symbol (@) is a placeholder for the trailing currency symbol. Thus, the format string **$#,###.##** will format the value 1234.56 as follows in a US English locale:

```
$1,234.56
```

It displays as follows in an French locale:

```
f1.234,56
```

When money values are converted to character strings using the LET statement, both the default conversion and the conversion with a USING clause insert the locale-specific separators and currency symbol into the created strings, not the US English separators and currency symbol.

The LC_MONETARY setting is impacted by settings in DBFORMAT, DBMONEY, and LANG, according to the hierarchy of precedence. The following diagram illustrates the precedence of environment variables in specifying monetary formatting:

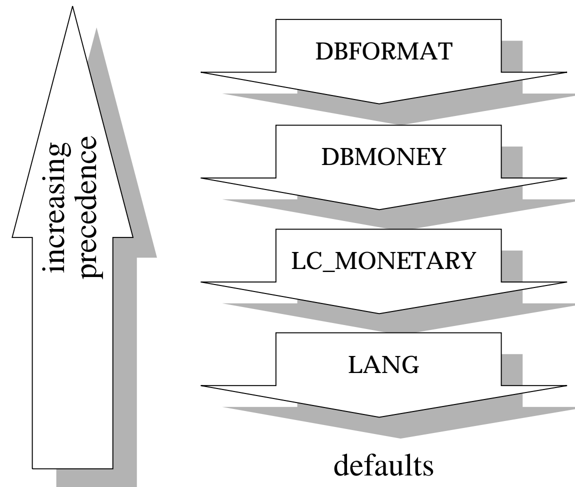


Figure E-10 Order of Precedence of Monetary Environment Variables

In the following statement, LC_MONETARY specifies the French currency symbol and format inherent in the ISO 8859/1 character set:

```
setenv LC_MONETARY FR_fr.88591
```

An example of printing a monetary value (without a USING clause) with German locale monetary formatting appears below. The statement:

```
    PRINT 1234.56
```

will produce the result:

```
1234,56DM
```

With the USING clause added, the statement:

```
    PRINT 1234.56 USING "#,###.##@@"
```

will print the result:

```
1.234,56DM
```

In **4GL**, there is a distinction between the interpretation by the database server of monetary values enclosed in quotes and those not enclosed in quotes. In **4GL**, unless values are contained in quotes, the database engine interprets all incoming data as being in ANSI SQL numeric data format (where the decimal separator is a period).

Note: *The use of LC_MONETARY for the specification of monetary formatting is preferable to the use of DBMONEY. DBMONEY is an obsolescent formatting construct that has been retained for backwards compatibility with older user-created programs. DBMONEY is also used by Informix engine tools that do not provide DBFORMAT. Note also that LC_MONETARY and LC_NUMERIC are the only means by which you can specify different formatting for monetary and numeric data. The formatting specified in DBFORMAT and DBMONEY apply to both monetary and numeric data. For more details, refer to the sections entitled "DBFORMAT" and "DBMONEY" in* *Appendix D, "Environment Variables."*

LC_NUMERIC

The X/Open-defined LC_NUMERIC environment variable sets the format for values of data types INTEGER, SMALLINT, DECIMAL, FLOAT, and SMALLFLOAT. This default format affects how numeric values are:

- Displayed and input on the screen
- Printed
- Input from ASCII files using LOAD
- Output to ASCII files using UNLOAD

LC_NUMERIC defines the numeric decimal and numeric thousands separators. Note that the decimal and thousands separators defined for numeric data by LC_NUMERIC are distinct from the decimal and thousands separators defined for monetary data by LC_MONETARY.

For example, the number 2345.67 in a US English locale displays as:

```
2,345.67
```

With LC_NUMERIC set for the French locale, where the thousands separator is the comma and the decimal separator the period, the value displays as:

```
2.345,67
```

The setting in LC_NUMERIC affects the following in 4GL:

- USING expression
- FORMAT attribute
- PRINT statement
- LET statement, where a character string is receiving a numeric value
- DISPLAY statement

The formal syntax for setting the LC_NUMERIC variable is as follows:

```
setenv LC_NUMERIC language ---------------------------------------------|
                             \_ _territory _/ \_ .charset _/ \_ @modifier _/
```

language a one to two character abbreviation such as *FR* for French, *DE* for German or *EN* for English.

_territory specifies a dialect of a language, such as *DE_ch* (Swiss dialect of German) or *EN_uk* (British dialect of English).

.charset specifies a character set other than ASCII, such as *.88591* (the ISO 8859/1 character set).

@modifier specifies a collation sequence within a language and territory, such as *@dictionary* (dictionary order) or *@telephone* (telephone book order).

In practice, the syntax for setting LC_NUMERIC is likely to be one of the following:

```
setenv LC_NUMERIC language
```

or

```
setenv LC_NUMERIC language_territory
```

There is no standardization of LC_ variable values between systems. Refer to the discussion of locale files on page E-26 for a better understanding of this syntax.

Usage

The setting in LC_NUMERIC determines the numeric and decimal separators. It changes the separators displayed on the screen in a numeric field and in the default format of a PRINT statement. For example, the value 1234.56 will print or display as follows in a French or German locale:

```
1234,56
```

In the case of a screen form, in the French or German locale values input by the user will be expected to contain commas, not periods, as decimal separators.

The setting in LC_NUMERIC also affects the way format strings in the FORMAT attribute and the USING clause are interpreted. In these format strings, the period symbol (.) is not a literal character but a placeholder for the decimal separator specified by environment variables. Likewise, the comma symbol (,) is a placeholder for the thousands separator specified by environment variables. Thus, the format string #,###.## will format the value 1234.56 as follows in a US English locale:

```
1,234.56
```

but as follows in a French or German locale:

```
1.234,56
```

When numeric values are converted to character strings using the LET statement, both the default conversion and the conversion with a USING clause will insert locale-specific separators into the created strings, not US English separators.

The LC_NUMERIC setting is impacted by settings in DBFORMAT and DBMONEY, according to the hierarchy of precedence. The following diagram illustrates the precedence of environment variables in specifying numeric formatting:

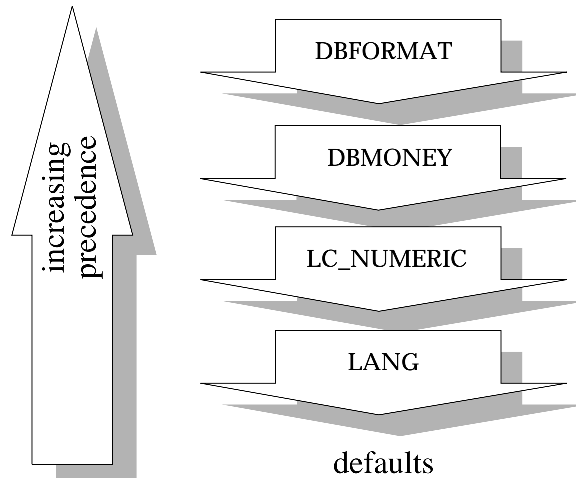


Figure E-11 Order of Precedence of Numeric Environment Variables

The following command sets the LC_NUMERIC environment variable to specify a variety of German numeric formatting:

```
setenv LC_NUMERIC DE_de.88591
```

Here, DE means *Deutsch* (German), de represents *Deutschland* (Germany), and **88591** represents the Western European version of the 8-bit ISO character set (ISO 88591:1983).

An example of printing a numeric value (without a USING clause) with German locale numeric formatting appears below. The statement:

```
    PRINT 1234.56
```

will produce the result:

```
1234,56
```

With the USING clause added, the statement:

```
PRINT 1234.56 USING "#,####.##"
```

will print the result:

```
1.234,56
```

In **4GL**, there is a distinction between the interpretation by the database server of numeric values enclosed in quotes, and those not enclosed in quotes. In **4GL**, unless values are contained in quotes, the database engine interprets all incoming data as being in ANSI SQL numeric data format (where the decimal separator is a period).

Informix-Defined Language and Formatting Variables

Informix-defined language and formatting variables, like the locale variables, specify the language and formatting environment of the user session. These Informix-defined variables interact with the locale variables according to the hierarchy of precedence.

DBFORMAT

The Informix-defined DBFORMAT environment variable specifies the default format in which the user enters and 4GL inputs, displays, or prints values of the following data types:

- DECIMAL
- FLOAT
- SMALLFLOAT
- INTEGER
- SMALLINT
- MONEY

The default format specified in DBFORMAT affects how numeric and monetary values are:

- Displayed and input on the screen
- Printed
- Input from ASCII files using LOAD
- Output to ASCII files using UNLOAD

DBFORMAT is used to specify the leading and trailing currency symbols and the decimal and thousands separators. DBFORMAT specifies the currency symbols, not their default positions within a monetary value. Note that the decimal and thousands separators defined by DBFORMAT apply to both monetary and numeric data, and override the sets of separators established by LC_MONETARY and LC_NUMERIC.

The setting in DBFORMAT affects the following in 4GL:

- USING expression
- FORMAT attribute
- PRINT statement
- LET statement, where a character string is receiving a monetary or numeric value
- DISPLAY statement

The DBFORMAT setting overrides settings in DBMONEY, LC_NUMERIC, and LC_MONETARY.

For a complete discussion of the DBFORMAT variable, refer to Appendix D, "Environment Variables."

Note: *The DBFORMAT variable, like DBMONEY and DBDATE, performs its role regardless of whether or not NLS is active (DBNLS set to 1 or 2). This is in contrast to LANG and the LC_ variables, which are only active when NLS is active.*

DBMONEY

The Informix-defined DBMONEY environment variable specifies the display format for MONEY values.

For example, the DBMONEY setting

,DM

prints the value 12345.67 as

```
12345,67DM
```

DBMONEY formats monetary data in a rough country-specific format, but provides no facility for a thousands separator. For complete information on DBMONEY, see Appendix D, "Environment Variables."

The precedence relationship between DBFORMAT, DBMONEY, and LC_MONETARY is illustrated below:

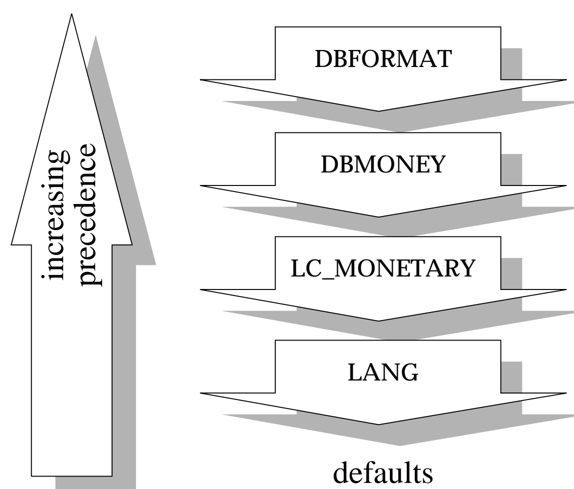


Figure E-12 Order of Precedence of Monetary Environment Variables

DBMONEY represents syntax from older versions of the product set. It is recommended that you use the LC_MONETARY environment variable for specifying monetary format. DBMONEY is retained only for compatibility with older versions.

The DBMONEY variable, like DBFORMAT and DBDATE, performs its role regardless of whether or not NLS is active (DBNLS set to 0 or 1). This is in contrast to LANG and the LC_ variables, which are only active when NLS is active.

DBDATE

Refer to Appendix D, "Environment Variables," for a complete discussion of DBDATE. The following points pertain to the relationship between DBDATE and NLS.

- DBDATE is insensitive to NLS locale, and to the effects of DBLANG. Since month and day are displayed as numeric values in all available DBDATE formats, they will not change with DBLANG the way formats containing character month names do.
- Date formatting specified in a USING clause or FORMAT attribute will override the formatting specified in DBDATE.
- The LANG setting will specify the default value for DBDATE in an active NLS environment on HP and IBM systems. On SUN systems LANG has no influence on the default for DBDATE.
- The DBDATE variable, like DBFORMAT and DBMONEY, performs its role regardless of whether or not NLS is active (DBNLS set to 1 or 2). This is in contrast to LANG and the LC_ variables, which are only active when NLS is active.

Note: *Although the XPG3 specification includes the ability to create locale-specific formatting of date and time data by way of the LC_TIME category, this is not supported in the 6.0 Informix tools.*

LOAD and UNLOAD Statements

NLS affects the text files produced by UNLOAD. It also determines the format in which it expects LOAD files. The UNLOAD statement uses the environment variables DBFORMAT, DBMONEY, LC_NUMERIC, LC_MONETARY, and DBDATE to determine its output format. The precedence of these format specifications is consistent with that of forms and reports. The LOAD statement expects incoming data in the format specified by the environment. If there is an inconsistency between the format of the data being loaded and the value of the formatting variables, an error is reported, and the LOAD is cancelled.

The order of precedence for monetary data is as illustrated in Figure E-10 on page E-34. The precedence order for numeric data is in Figure E-11 on page E-38. Date data is currently only affected by DBDATE.

UNLOAD utilizes the user's language and formatting environment information as follows:

- For dates, as specified in DBDATE.
- For numbers, as specified in DBFORMAT, DBMONEY, and LC_NUMERIC (although not allowing thousands separators).
- For money values, as specified in DBFORMAT, DBMONEY, and LC_MONETARY (although not allowing thousands separators).

LOAD uses the user's environment information as follows:

- For dates, as specified in DBDATE.
- For numbers, as specified in DBFORMAT, DBMONEY, and LC_NUMERIC (including thousands separators).
- For money values, as specified in DBFORMAT, DBMONEY, and LC_MONETARY including thousands separators and currency symbols, but not following the usual negative numbering conventions, that is, (-) = negative.

This set of behaviors is adequate for unloading and loading between two non-NLS databases, or between two databases with the same locale. For unloading and loading between dissimilar database locales, it is necessary to perform one of the following two sequences of operations:

- At the source database, set DBNLS to 2 and LANG to the locale of the destination database, then perform the UNLOAD operation. LOAD the resulting text file at the destination database using normal settings for that database.

  or

- At the source database, UNLOAD using normal settings. At the destination database, set DBNLS to 2 and LANG to the locale of the source database, and perform the LOAD.

Either of these approaches works equally well. An example of the first approach would be a user in Paris unloading a database created with LANG=FR, and loading the data in Munich to a database created with LANG=DE. The user in Paris sets DBNLS to 2 and LANG to DE at the client side before performing the unload. The data is sent to Munich, where the user there performs the load normally (that is, with DBNLS set to 1 and consistent locale variables).

Note that if cross-locale database load/unload sequences take place without following one of the two procedures identified above, the following can occur:

- Multiplying or dividing numeric or monetary values by orders of magnitude if thousands and decimal separators are different between locales. For example, loading an English UNLOAD file into a German database causes every value to be divided by 1000, because of the switched roles of the comma and period symbols.
- Generating LOAD errors if characters are encountered that are outside the character set of the database.
- Generating LOAD errors if uninterpretable currency symbols are encountered.

FORMAT and USING

The USING operator is typically used in DISPLAY or PRINT statements, but you can also use it with LET to assign the formatted value to a character variable. The FORMAT attribute is used strictly in screen forms. Format strings in USING and FORMAT have identical meanings, except that a FORMAT operator's formatting string cannot display currency symbols, whereas USING format strings can. NLS variables influence the results obtained from USING and FORMAT strings identically, except for currency symbols not being available in FORMAT. The order of variable precedence is as illustrated in the sections on LC_MONETARY and LC_NUMERIC.

The following classifications apply to format strings:

- A format string is said to be a **monetary formatter** if either one of the following two conditions hold:
  - The format string is formatting monetary data.
  - The format string contains either of the currency placeholder symbols (**$** or **@**).

  Format strings in the USING operator can be monetary formatters. Format strings in the FORMAT attribute cannot.
- A format string is said to be a **date formatter** if it is not a monetary formatter and it contains one of the following tokens: ***mm***, ***mmm***, ***dd***, ***ddd***, or *yy*, *yyyy*.
- A format string is said to be a **numeric formatter** if it is neither a monetary formatter nor a date formatter.

Monetary and Numeric Formatting

The following format string symbols have new meanings in 6.0 4GL. They are influenced by settings in the LC_MONETARY, LC_NUMERIC, DBFORMAT, DBMONEY, DBDATE, and DBLANG environment variables. Any valid formatting symbol that is not mentioned below has the same meaning it had in the pre-6.0 4GL.

, Any appearance of a comma in the format string for non-DATE data is interpreted as a thousands separator. This symbol is not a literal. The comma stands for either the numeric thousands separator or the monetary thousands separator specified by the format environment. If the format string is a monetary-formatter, then this symbol stands for the monetary thousands separator. If the format string is a numeric-formatter, then this symbol stands for the numeric thousands separator. If the format string is a date-formatter, then this symbol stands for itself; the comma remains a comma.

. Any appearance of a period in the format string for non-DATE data is interpreted as a decimal separator. This symbol is not a literal. The period stands for the decimal separator specified by the format environment. If the format string is a monetary-formatter, then this symbol stands for the monetary decimal separator. If the format string is a numeric-formatter, then this symbol stands for the numeric decimal separator. If the format string is a date-formatter, then this symbol stands for itself; the period remains a period.

$ In a USING operator format string, the dollar sign is the leading currency formatting symbol. The dollar sign is not valid syntax for the FORMAT attribute, because the FORMAT attribute cannot format the MONEY data type. Groups of dollar signs stand for formatting space provided for the currency symbols that *precede* a monetary value. A group of dollar signs in a row will provide formatting space for the leading currency symbol or string specified in the format environment. The USING clause right-justifies currency symbols within the sequence of **$** signs.

@ In a USING operator format string, the at symbol (@) is the trailing currency symbol formatting symbol. This symbol, which is new to USING format strings, is similar in purpose to the dollar sign. Groups of **@** symbols stand for formatting space provided for the currency symbol or string that *follows* a monetary value. The USING clause left-justifies currency symbols within the sequence of **@** signs.

For example, with LANG set to DE, and columns **m** and **c** representing monetary and character data, respectively, a German **4GL** program might use the following statements to format money:

```
LET m = 1234.56
LET c = m USING "#,###.##@@"
PRINT c
```

The result would appear as shown:

```
1.234,56DM
```

With LANG set to EN_us and DBFORMAT set to:

```
£:,:.:p
```

a British **4GL** program might use the following statements to format money:

```
LET m = 1234.56
LET c = m USING "$$#,###.##@"
PRINT c
```

The result would be as follows:

```
£1,234.56p
```

Note the use of both a leading and a trailing currency symbol in the latter example.

Date Formatting

If you use the USING operator or FORMAT attribute to format a DATE value, USING or FORMAT takes precedence over the DBDATE environment variable. The format-string for a date can be a combination of the characters *m*, *d*, and *y*:

dd	day of the month as a 2-digit number (01 through 31 or less)
ddd	day of the week as a 3-letter abbreviation (Sun through Sat)
mm	month as a 2-digit number (01 through 12)
mmm	month as a 3-letter abbreviation (Jan through Dec)
yy	year as a 2-digit number in the 1900s (00 through 99)
yyyy	year as a 4-digit number (0001 through 9999)

The DBLANG setting and installation of appropriate message files will determine the set of available 3-character weekday name ***ddd*** and month name ***mmm*** abbreviations.

DISPLAY Statements

When 4GL displays a number value, it follows these rules:

- Displays the leading currency symbol (as set by DBFORMAT or DBMONEY) for MONEY values. If the FORMAT attribute specifies a leading currency symbol for other data types, then 4GL displays that symbol.
- Omits the thousands separator, unless it is specified by a FORMAT attribute or by the USING operator.
- Displays the decimal separator, except for INT or SMALLINT values.
- Displays the trailing currency symbol (as set by DBFORMAT or DBMONEY) for MONEY values, unless you specify a FORMAT attribute or the USING operator. In this case, 4GL ignores the trailing currency symbol; the user cannot enter a trailing currency symbol, and 4GL does not display it.

LET Statements

When LET statements are employed without USING clauses, certain default conversions are employed in an NLS environment. In particular, there is the case where a monetary or numeric value is converted to a character value (or the character variable is decoded into monetary or a numeric value). In this situation, the conversion follows the formatting rules established by the

relevant hierarchy of environment variables. The following example (in which columns **c** and **c2** are of CHARACTER type, **m** is MONEY and **d** is DECIMAL) demonstrates this:

```
LET m = 1234.56; LET d = 9876.54
LET c = m; LET c2 = d
PRINT c, c2
```

The result in a LANG=DE environment is:

```
1.234,56DM  9876,54
```

When LET statements assign monetary and numeric constants to variables and NLS is active, the interpretation of the constants is governed by locale if the constant is in quotes, and governed by US ANSI if the constant is not in quotes. The following example (in which **m** and **m2** are of type MONEY) illustrates this:

```
DEFINE m1 MONEY, m2 MONEY
LET m1 = 1234.56; LET m2 = "1234,56DM"
```

In a LANG=DE environment, this example would work correctly and compile without error in **4GL**. However, in a LANG=EN_us (US English) environment, the constant being assigned to **m2** would trigger a compilation error.

Note: *For a user performing data entry into a form, locale is active for values being entered, and quotes are not necessary.*

Multiple Locale Support

Under version 6.0, an NLS database is limited to one locale per database. The database locale is specified by the contents of the LC_COLLATE and LC_CTYPE environment variables stored in the database at database creation time. However, multiple databases built with different environment variable settings can reside on the same server. Furthermore, multiple users can access databases with different locales on the same server at the same time. The locale of the server machine itself is irrelevant for the purposes of determining whether or not access will be permitted; it is strictly determined by the consistency checking process between user and database as described earlier.

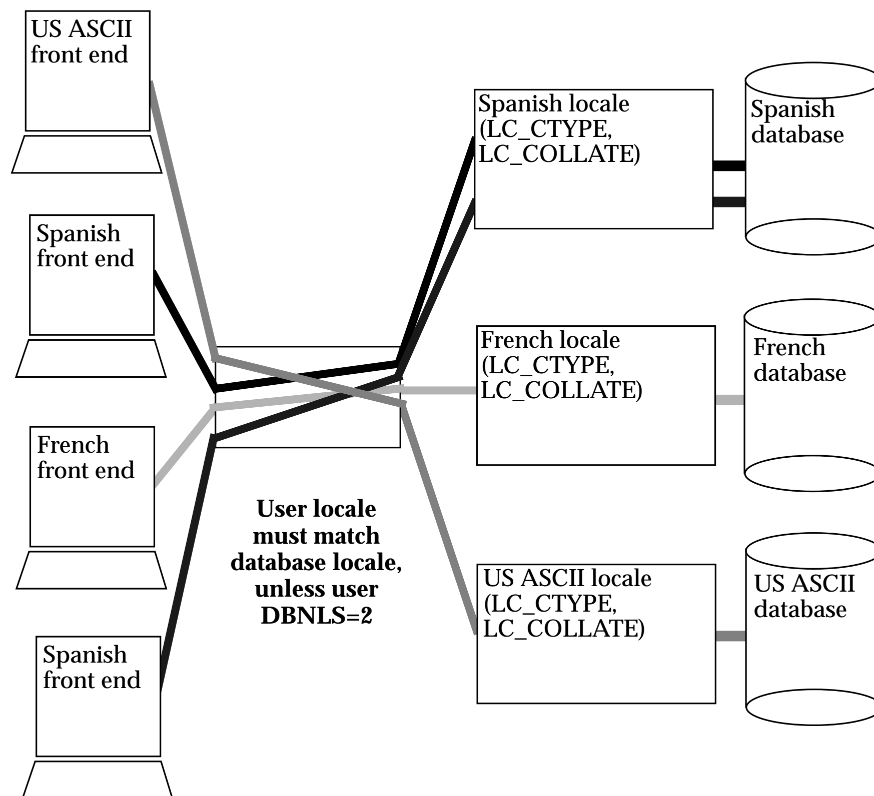


Figure E-13 Multiple NLS Locales in 6.0 Architecture

Language Supplements

With NLS, in order to use different languages, you need the necessary language supplements to install on the system. For example, to use German and Spanish with the 4GL product, you need to buy an 4GL package and two language supplements: one German and one Spanish. The installation script for the language supplements creates the appropriate directories.

In order to know the exact language supplement to purchase, you need to know the following information:

- The system (Sun, HP, and so on)
- Operating system version
- NLS version (if purchased separately)
- Information about your locale (LANG setting)

Contact your local Informix sales office for more information on language supplements.

Appendix

Modifying *termcap* and *terminfo*

INFORMIX-4GL programs can use function keys and can display color or intensity attributes in screen displays. These and other keyboard and screen options are terminal dependent. To determine terminal-dependent characteristics, **INFORMIX-4GL** uses the information in the **termcap** file or in the **terminfo** directory. **INFORMIX-4GL** uses the INFORMIXTERM environment variable to determine whether to use **termcap** or **terminfo**. For more information about INFORMIXTERM, read the discussion of environment variables in Appendix D, "Environment Variables."

With 4GL, Informix distributes **termcap** files that contain additional capabilities for many common terminals (such as the Wyse 50 and the Televideo 950). These capabilities include intensity-change or color-change descriptions or both. This appendix describes these capabilities, as well as the general format of **termcap** and **terminfo** entries.

Since **terminfo** does not support color, you can only use **INFORMIX-4GL** color functionality with **termcap**. If you want to use color in **INFORMIX-4GL**, you must set the INFORMIXTERM environment variable to **termcap**.

You can use the information in this appendix, combined with the information in your terminal manual, to modify the contents of your **termcap** file or **terminfo** files. This appendix is divided into two main sections, **termcap** and **terminfo**. Depending on which you are using, you should read the appropriate section.

termcap

When **INFORMIX-4GL** is installed on your system, a **termcap** file is placed in **$INFORMIXDIR/etc**. This file is a superset of an operating system **termcap** file. The Informix **termcap** file contains additional capabilities for many terminals. You may want to modify this file further in the following instances:

- The entry for your terminal has not been modified to include color-change and intensity-change capabilities.
- You want to extend function key definitions.
- You want to specify or alter the graphics characters used for window border.
- You want to customize your terminal entry in other ways.

Note: *Some terminals cannot support color or graphics characters. You should read this appendix and the user guide that comes with your terminal to determine whether or not the changes described in this appendix are applicable to your terminal.*

Format of a *termcap* Definition

This section describes the general format of **termcap** entries. For a complete description of **termcap**, refer to your operating system documentation.

A **termcap** entry contains a list of names for the terminal, followed by a list of the terminal's capabilities. There are three types of capabilities:

- Boolean capabilities
- Numeric capabilities
- String capabilities

All **termcap** entries have the following format:

- Is specified as a backslash (\ ) followed by the letter E, and CONTROL is specified as a caret (^). Do not use the ESCAPE or CONTROL keys to indicate escape sequences or control characters in a **termcap** entry.
- Each capability, including the last one in the entry, is followed by a colon (:).
- Entries must be defined on a single logical line; a backslash (\ ) appears at the end of each line that wraps to the next line.

Figure F-1 shows a basic **termcap** entry for the Wyse 50 terminal:

```
# Entry for Wyse 50:

w5|wy50|wyse50:
:if=/usr/lib/tabset/std:\
:al=\EE:am:bs:ce=\Et:cm=\E=%+ %+ :cl=\E*:co#80:\
:dc=\EW:dl=\ER:ho=^^:ei=:kh=^^:im=:ic=\EQ:in:li#24:\
:nd=^L:pt:se=\EG0:so=\EG4:sg#1:ug#1:\
:up=^K:ku=^K:kd=^J:kl=^H:kr=^L:kb=:\
:k0=^A@^M:k1=^AA^M:k2=^AB^M:k3=^AC^M:k4=^AD^M:\
:k5=^AE^M:k6=^AF^M:k7=^AG^M:\
:HI=^|:Po=^R:Pe=^T:
```

Figure F-1 ***Wyse 50* termcap *Entry***

Note: *Comment lines begin with a pound sign (#).*

Terminal Names

A **termcap** entry starts with one or more names for the terminal, each separated by a vertical (|) bar. For example, the **termcap** entry for the Wyse 50 terminal starts with the following line:

```
w5|wy50|wyse50:\
```

The **termcap** entry can be accessed using any one of these names.

Boolean Capabilities

A Boolean capability is a two-character code that indicates whether or not a terminal has a specific feature. If the Boolean capability is present in the **termcap** entry, the terminal has that particular feature. Figure F-2 shows some of the Boolean capabilities for the Wyse 50 terminal:

```
     :bs:am:

#   bs   backspace with CTRL-H
#   am   automatic margins
```

Figure F-2 ***Boolean Capabilities for the Wyse 50***

Numeric Capabilities

A numeric capability is a two-character code followed by a pound symbol (#) and a value. Figure F-3 shows the numeric capabilities for the number of columns and the number of lines on a Wyse 50 terminal:

```
     :co#80:li#24:

#   co   number of columns in a line
#   li   number of lines on the screen
```

Figure F-3 ***Numeric Capabilities for the Wyse 50***

Similarly, **sg** is a numeric capability that indicates the number of character positions required on the screen for reverse video. The entry `:sg#1:` indicates that a terminal requires one additional character position when reverse video is turned ON or OFF. If you do not include a particular numeric capability, **INFORMIX-4GL** assumes that the value is zero.

String Capabilities

A string capability specifies a sequence that can be used to perform a terminal operation. A string capability is a two-character code followed by an equal sign (=) and a string ending at the next delimiter (:).

Most **termcap** entries include string capabilities for clearing the screen, cursor movement, Arrow keys, underscore, function keys, and so on. Figure F-4 shows many of the string capabilities for the Wyse 50 terminal:

```
:ce=\Et:cl=\E*:\
:nd=^L:up=^K:\
:so=\EG4:se=\EG0:\
:ku=^K:kd=^J:kr=^L:kl=^H:\
:k0=^A@^M:k1=^AA^M:k2=^AB^M:k3=^AC^M:


#    ce=\Et          clear to end of line
#    cl=\E*          clear the screen
#    nd=^L           non-destructive cursor right
#    up=^K           up one line
#
#    so=\EG4         start stand-out
#    se=\EG0         end stand-out
#
#    ku=^K           up arrow key
#    kd=^J           down arrow key
#    kr=^L           right arrow key
#    kl=^H           left arrow key
#
#    k0=^A@^M        function key F1
#    k1=^AA^M        function key F2
#    k2=^AB^M        function key F3
#    k3=^AC^M        function key F4
```

Figure F-4 String Capabilities for the Wyse 50

Extending Function Key Definitions

INFORMIX-4GL recognizes function keys **F1** through **F36**. These keys correspond to the **termcap** capabilities **k0** through **k9**, followed by **kA** through **kZ**. The **termcap** entry for these capabilities is the sequence of ASCII characters your terminal sends when you press the function keys (or any other keys you choose to use as function keys). For the Wyse 50 and Televideo 950 terminals, the first eight function keys send the characters shown in Figure F-5.

Function Key	termcap Entry
F1	k0=^A@^M
F2	k1=^AA^M
F3	k2=^AB^M
F4	k3=^AC^M
F5	k4=^AD^M
F6	k5=^AE^M
F7	k6=^AF^M
F8	k7=^AG^M

Figure F-5 Function Key Entries for the Wyse 50

You can also define keys that correspond to the following capabilities:

- Insert line (**ki**)
- Delete line (**kj**)
- Next page (**kf**)
- Previous page (**kg**)

If these keys are defined in your **termcap** file, **INFORMIX-4GL** uses them. Otherwise, **INFORMIX-4GL** uses CONTROL-J, CONTROL-K, CONTROL-M, and CONTROL-N, respectively.

Note: *You can also use the OPTIONS statement to name other function keys or CONTROL keys for these operations.*

Specifying Characters for Window Borders

INFORMIX-4GL uses characters defined in the **termcap** file to draw the border of a window. If no characters are defined in this file, **INFORMIX-4GL** uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners.

The **termcap** file provided with **INFORMIX-4GL** contains border character definitions for many common terminals. You can look at the **termcap** file to see if the entry for your terminal has been modified to include these definitions. If your terminal entry does not contain border character definitions, or if you want to specify alternative border characters, you or your system administrator can modify the **termcap** file.

Perform the following steps to modify the definition for your terminal type in the **termcap** file:

1. Determine the escape sequences for turning graphics mode ON and OFF. This information is located in the manual that comes with your terminal.

For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B and the escape sequence for leaving graphics mode is ESC H^C.

Note: *Terminals without a graphics mode do not have this escape sequence. The procedure for specifying alternative border characters on a non-graphics terminal is discussed at the end of this section.*

2. Identify the ASCII equivalents for the six graphics characters that **INFORMIX-4GL** requires to draw the border. (The ASCII equivalent of a graphics character is the key you would press in graphics mode to obtain the indicated character.)

   Figure F-6 shows the graphics characters and the ASCII equivalents for a Wyse 50 terminal.

Window Border Position	Graphics Character	ASCII Equivalent
upper left corner	⌈	2
lower left corner	⌊	1
upper right corner	⌉	3
lower right corner	⌋	5
horizontal	-	z
vertical	|	6

Figure F-6 ***Wyse 50 ASCII Equivalents for Border Graphics Characters***

   Again, this information should be located in the manual that comes with your terminal.

3. Edit the **termcap** entry for your terminal.

   Note: *You may want to make a copy of your* ***termcap*** *file before you edit it. You can use the TERMCAP environment variable to point to whichever copy of the* ***termcap*** *file you want to access.*

   Use the format:

   termcap-capability=value

   to enter values for the following **termcap** capabilities:

   gs The escape sequence for entering graphics mode. In the **termcap** file, is represented as a backslash (\ ) followed by the letter E;

CONTROL is represented as a caret (^). For example, the Wyse 50 escape sequence ESC-H CONTROL-B is represented as \EH^B.

ge The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESC-H CONTROL-C is represented as \EH^C.

gb The concatenated, ordered list of ASCII equivalents for the six graphics characters used to draw the border. Use the following order:

upper left corner
lower left corner
upper right corner
lower right corner
horizontal lines
vertical lines

Follow these guidelines when you insert information in the **termcap** entry:

1. Delimit entries with a colon (:).
2. End each continuing line with a backslash (\ ).
3. End the last line in the entry with a colon.

For example, if you are using a Wyse 50 terminal, you would add the following information in the **termcap** entry for the Wyse 50:

```
:gs=\EH^B:\       # sets gs to ESC H CTRL B
:ge=\EH^C:\       # sets ge to ESC H CTRL C
:gb=2135z6:\      # sets gb to the ASCII equivalents
                  # of graphics characters for upper
                  # left, lower left, upper right,
                  # lower right, horizontal,
                  # and vertical
```

If you prefer, you can enter this information in a linear sequence.

```
:gs=\EH^B:ge=\EH^C:gb=2135z6:\
```

If Your termcap File Contains sg#1 Capabilities

The **termcap** file for some terminals contains sg#1 capabilities. If sg#1 is included, 4GL reserves an additional column to the left and right of the window. If you specify a border around the 4GL window, these two columns are in addition to the two additional columns required for the border.

Terminals Without Graphics Capabilities

For terminals without graphics capabilities, you must enter a blank value for the **gs** and **ge** capabilities. For **gb**, enter the characters you want **INFORMIX-4GL** to use for the window border.

The following example shows possible values for **gs**, **ge**, and **gb** in an entry for a terminal without graphics capabilities. In this example, window borders would be drawn using underscores (_) for horizontal lines, vertical bars (|) for vertical lines, periods (.) for the top corners, and vertical bars (|) for the lower corners.

```
:gs=:ge=:gb=.|.|_|:
```

INFORMIX-4GL uses the graphics characters in the **termcap** file when you specify a window border in an OPEN WINDOW statement.

Adding Color and Intensity

Many of the terminal entries in the Informix **termcap** file have been modified to include color or intensity capabilities or both. (The **termcap** file is located in the **$INFORMIXDIR/etc** directory.) You can view the **termcap** file to determine if the entry for your terminal type includes these capabilities. If your terminal entry includes the **ZA** capability, your terminal is set up for color or intensity or both. If it doesn't, you can add color and intensity capabilities by using the information in this section. The following topics are outlined in this section:

- Color and intensity
- The **ZA** capability
- Stack operations
- Examples

You should understand these topics before you modify your terminal entry.

Color and Intensity Attributes

You can write your **INFORMIX-4GL** program either for a monochrome or a color terminal and then run the program on either type of terminal. If you set up the **termcap** files as described here, the color attributes and the intensity attributes are related, as shown in Figure F-7.

Number	Color Terminal	Monochrome Terminal
0	WHITE	NORMAL
1	YELLOW	BOLD
2	MAGENTA	BOLD
3	RED	BOLD†
4	CYAN	DIM
5	GREEN	DIM
6	BLUE	DIM†
7	BLACK	DIM

Figure F-7 Color-Monochrome Correspondence

The background for colors is BLACK in all cases. In the Figure F-7, the † signifies that, if the keyword BOLD is indicated as the attribute, the field will be RED on a color terminal, or if the keyword DIM is indicated as the attribute, the field will be BLUE on a color terminal.

You can change the color names from the default list by associating different numbers with different color names in a file named **colornames** in your current directory or in the **$INFORMIXDIR/incl** directory. (See the section "The colornames File" on page F-19.)

In either color or monochrome mode, you can add the REVERSE, BLINK, or UNDERLINE attributes if your terminal supports them. You can select only one of these three attributes.

The ZA String Capability

INFORMIX-4GL uses a parameterized string capability **ZA** in the **termcap** file to determine color assignments. Unlike other **termcap** string capabilities that you set equal to a literal sequence of ASCII characters, **ZA** is a function string that depends upon four parameters:

Parameter 1 (p1)Color number between 0 and 7 (see Figure F-7)
Parameter 2 (p2)0 = Normal; 1 = Reverse
Parameter 3 (p3)0 = No-Blink; 1 = Blink
Parameter 4 (p4)0 = No-Underscore; 1 = Underscore

ZA uses the values of these four parameters and a stack machine to determine which characters to send to the terminal. The **ZA** function is called and these parameters are evaluated when a color or intensity attribute is encountered in a **4GL** program. You can use the information in your terminal manual to set the **ZA** parameters to the correct values for your terminal.

To define the **ZA** string for your terminal, you use *stack operators* to push and pop values onto and off the *stack*. The next section describes several stack operators. Use these descriptions and the subsequent examples to understand how to define the string for your terminal.

Stack Operations

The **ZA** string uses stack operations to either push values onto the stack or pop values off the stack. Typically, the instructions in the **ZA** string push a parameter onto the stack, compare it to one or more constants, and then send an appropriate sequence of characters to the terminal. More complex operations are often necessary and, by storing the display attributes in static stack machine registers (named **a** through **z**), you can achieve terminal-specific optimizations.

A summary follows of the different stack operators you can use to write the descriptions. For a complete discussion of stack operators, consult your operating system documentation.

Operators that Send Characters to the Terminal

%d pops a numeric value from the stack and sends a maximum of three digits to the terminal. For example, if the value `145` is at the top of the stack, `%d` pops the value off the stack and sends the ASCII representations of 1, 4, and 5 to the terminal. If the value `2005` is at the top of the stack, `%d` pops the value off the stack and sends the ASCII representation of 5 to the terminal.

%2d pops a numeric value from the stack and sends a maximum of two digits to the terminal, padding to two places. For example, if the value `145` is at the top of the stack, `%2d` pops the value off the stack and sends the ASCII representations of 4 and 5 to the terminal. If the value `5` is at the top of the stack, `%2d` pops the value off the stack and sends the ASCII representations of 0 and 5 to the terminal.

%3d pops a numeric value from the stack and sends a maximum of three digits to the terminal, padding to three places. For example, if the value `7` is at the top of the stack, `%3d` pops the value off the stack and sends the ASCII representations of 0, 0, and 7 to the terminal.

%c pops a single character from the stack and sends it to the terminal.

Operators that Manipulate the Stack

%p[1-9] pushes the value of the specified parameter on the stack. The notation for parameters is p1, p2, ... p9. For example, if the value of p1 is `3`, `%p1` pushes `3` on the stack.

%P[a-z] pops a value from the stack and stores it in the specified variable. The notation for variables is Pa, Pb, ... Pz. For example, if the value `45` is on the top of the stack, `%Pb` pops `45` from the stack and stores it in the variable Pb.

%g[a-z] gets the value stored in the corresponding variable (P[a-z]) and pushes it on the stack. For example, if the value `45` is stored in the variable Pb, `%gb` gets 45 from Pb and pushes it on the stack.

%´c´ pushes a single character on the stack. For example, `%´k´` pushes `k` on the stack.

%{n} pushes an integer constant on the stack. The integer can be any length and can be either positive or negative. For example, `%{0}` pushes the value `0` on the stack.

%S[a-z] pops a value from the stack and stores it in the specified static variable. (Static storage is nonvolatile since the stored value remains from one attribute evaluation to the next.) The notation for static variables is Sa, Sb, ... Sz. For example, if the value `45` is on the top of the stack, `%Sb` pops `45` from the stack and stores it in the static variable Sb. This value is accessible for the duration of the **INFORMIX-4GL** program.

%G[a-z] gets the value stored in the corresponding static variable (S[a-z]) and pushes it on the stack. For example, if the value `45` is stored in the variable Sb, `%Gb` gets `45` from Sb and pushes it on the stack.

Arithmetic Operators

Each arithmetic operator pops the top two values from the stack, performs an operation, and pushes the result on the stack.

%+ Addition. For example, `%{2}%{3}%+` is equivalent to 2+3.

%- Subtraction. For example, `%{7}%{3}%-` is equivalent to 7-3.

%* Multiplication. For example, `%{6}%{3}%*` is equivalent to 6*3.

%/ Integer division. For example, `%{7}%{3}%/` is equivalent to 7/3 and produces a result of 2.

%m Modulus (or remainder). For example, `%{7}%{3}%m` is equivalent to (7 mod 3) and produces a result of 1.

Bit Operators

The following bit operators pop the top two values from the stack, perform an operation, and push the result on the stack:

%& Bit-and. For example, `%{12}%{21}%&` is equivalent to (12 and 21) and produces a result of 4.

```
        Binary              Decimal

   0   1   1   0   0       =    12

   1   0   1   0   1       =    21

-------------------------        and

   0   0   1   0   0       =    4
```

%| Bit-or. For example, `%{12}%{21}%|` is equivalent to (12 or 21) and produces a result of 29.

```
        Binary              Decimal

   0   1   1   0   0       =    12

   1   0   1   0   1       =    21

-------------------------       or

   1   1   1   0   1       =    29
```

%^ Exclusive-or. For example, `%{12}%{21}%^` is equivalent to (12 exclusive-or 21) and produces a result of 25.

```
        Binary              Decimal

   0   1   1   0   0       =    12

   1   0   1   0   1       =    21

-------------------------       exclusive or

   1   1   0   0   1       =    25
```

The following unary operator pops the top value from the stack, performs an operation, and pushes the result on the stack:

%~ Bitwise complement. For example, `%{25}%~` results in a value of -26, as shown in the following display.

```
          Binary                    Decimal

 0  0  0  1   1   0   0   1      =    25

----------------------------        Complement

 1  1  1  0   0   1   1   0      =   -26
```

Logical Operators

The following logical operators pop the top two values from the stack, perform an operation, and push the logical result (either `0` for false or `1` for true) on the stack:

%= Equal to. For example, if the parameter p1 has the value 3, the expression `%p1%{2}%=` is equivalent to 3=2 and produces a result of 0 (false).

%> Greater than. For example, if the parameter p1 has the value 3, the expression `%p1%{0}%>` is equivalent to 3>0 and produces a result of 1 (true).

%< Less than. For example, if the parameter p1 has the value 3, the expression `%p1%{4}%<` is equivalent to 3<4 and produces a result of 1 (true).

The following unary operator pops the top value from the stack, performs an operation, and pushes the logical result (either 0 or 1) on the stack.

%! Logical negation. This operator produces a value of zero for all nonzero numbers and a value of `1` for zero. For example, `%{2}%!` results in a value of `0`, and `%{0}%!` results in a value of `1`.

Conditional Statements

The condition statement IF-THEN-ELSE has the following format:

%? ***expr*** %t ***thenpart*** %e ***elsepart*** %;

The %e ***elsepart*** is optional. You can nest conditional statements in the ***thenpart*** or the ***elsepart.***

When **INFORMIX-4GL** evaluates a conditional statement, it pops the top value from the stack and evaluates it as either `true` or `false`. If the value is `true`, **INFORMIX-4GL** performs the operations after the %t; otherwise it performs the operations after the %e (if any).

For example, the expression:

```
%?%p1%{3}%=%t;31%;
```

is equivalent to:

```
if p1 = 3 then print ";31"
```

Assuming that p1 has the value `3`, **INFORMIX-4GL** performs the following steps:

- %? does not perform an operation but is included to make the conditional statement easier to read.
- %p1 pushes the value of p1 on the stack.
- %{3} pushes the value `3` on the stack.
- %= pops the value of p1 and the value `3` from the stack, evaluates the Boolean expression p1=3, and pushes the resulting value `1 (true)` on the stack.
- %t pops the value from the stack, evaluates `1` as `true`, and executes the operations after %t. (Since ";31" is not a stack machine operation, **INFORMIX-4GL** prints ";31" to the terminal.)
- %; terminates the conditional statement.

Summary of Operators

Figure F-8 summarizes the allowed operations:

Operation	Description
%d	write pop() in decimal format
%2d	write pop() in 2-place decimal format
%3d	write pop() in 3-place decimal format
%c	write pop() as a single character
%p[1-9]	push *i*th parameter
%P[a-z]	pop and store variable
%g[a-z]	get variable and push on stack
%'c'	push char constant
%{n}	push integer constant
%S[a-z]	pop and store static variable
%G[a-z]	get static variable and push
%+	addition. push(pop() op pop())
%-	subtraction. push(pop() op pop())
%*	multiplication. push(pop() op pop())
%/	integer division. push(pop() op pop())
%m	modulus. push(pop() op pop())
%&	bit and. push(pop() op pop())
%|	bit or. push(pop() op pop())
%^	bit exclusive or. push(pop() op pop())
%~	bitwise complement. push(op pop())
%=	equal to. push(pop() op pop())
%>	greater than. push(pop() op pop())
%<	less than. push(pop() op pop())
%!	logical negation. push(op pop())
%?	*expr* %t *thenpart* %e *elsepart* %; if-then-else; the %e *elsepart* is optional. else-if's are possible (c's are conditions): %? c1 %t...%e c2 %t...%e c3 %t...%e...%; nested if's allowed.

all other characters are written to the terminal;
use '%%' to write '%'.

Figure F-8 Stack Operations

Examples

To illustrate, consider the monochrome Wyse terminal. Figure F-9 shows the escape sequences for various display characteristics.

Escape Sequence	Results
ESC G 0	Normal
ESC G 1	blank(invisible)
ESC G 2	blink
ESC G 4	Reverse
ESC G 5	Reverse and blank
ESC G 6	Reverse and blink
ESC G 8	Underscore
ESC G 9	Underscore and blank
ESC G :	Underscore and blink
ESC G <	Underscore and reverse
ESC G =	Underscore, reverse, and blank
ESC G >	Underscore, reverse, and blink

Figure F-9 Wyse Escape Sequences

The characters after `G` form an ASCII sequence from the character `0` (zero) through `?`. You can generate the character by starting with `0` and adding `1` for blank, `2` for blink, `4` for reverse, and `8` for underline.

You can construct the **termcap** entry in stages, as outlined in the following display. `%pi` refers to pushing the ith parameter on the stack. The designation for is `\E`. The **termcap** entry for the Wyse terminal must contain the following **ZA** entry in order for **INFORMIX-4GL** monochrome attributes such as REVERSE and BOLD to work correctly:

```
ZA =
EG                        #print EG
%'0'                      #push '0' (normal) on the stack
%?%p1%{7}%=%t%{1}%|       #if p1 = 7 (invisible), set
                          #the 1 bit (blank);
%e%p1%{3}%>               #if p1 > 3 and < 7, set the 64 flag (dim);
  %p1%{7}%<%&%t%{64}%|    #
     %;%;                 #
%?%p2%t%{4}%|%;           #if p2 is set, set the 4 bit (reverse)
%?%p3%t%{2}%|%;           #if p3 is set, set the 2 bit (blink)
%?%p4%t%{8}%|%;           #if p4 is set, set the 8 bit (underline)
%c:                       #print whatever character
                          #is on top of the stack
```

You then concatenate these lines as a single string that ends with a colon and has no embedded NEWLINEs. The actual **ZA** entry for the Wyse 50 terminal follows:

```
ZA = \EG%'0'%?%p1%{7}%=%t%{1}%|%e%p1%{3}%>%p1%{7}%<%&%t%{64}
%|%;%;%?%p2%t%{4}%|%;%?%p3%t%{2}%|%;%?%p4%t%{8}%|%;%c:
```

The next example is for the ID Systems Corporation ID231, a color terminal. On this terminal, to set color and other characteristics you must enclose a character sequence between a lead-in sequence (`ESC [ 0`) and a terminating character (`m`). The first in the sequence is a two-digit number that determines whether the assigned color is in the background (30) or in the foreground (40). The next is another two-digit number that is the other of 30 or 40, incremented by the color number. These characters are followed by 5 if there is blinking, and by 4 for underlining.

The code in Figure F-10 sets up the entire escape sequence:

```
ZA =
\E[0;                    #print lead-in
%?%p1%{0}%=%t%{7}        #encode color number (translate
%e%p1%{1}%=%t%{3}        #   from Figure F-7 to the number
%e%p1%{2}%=%t%{5}        #   for the ID231)
%e%p1%{3}%=%t%{1}        #
%e%p1%{4}%=%t%{6}        #
%e%p1%{5}%=%t%{2}        #
%e%p1%{6}%=%t%{4}        #
%e%p1%{7}%=%t%{0}%;      #
%?%p2%t30;%{40}%+%2d     #if p2 is set, print '30' and
                         #  '40' + color number (reverse)
%e40;%{30}%+%2d%;        #  else print '40' and
                         #  '30' + color number (normal)
%?%p3%t;5%;              #if p3 is set, print 5 (blink)
%?%p4%t;4%;              #if p4 is set, print 4 (underline)
m                        #print 'm' to end character
                         #  sequence
```

Figure F-10 Sample ZA String for ID231

When you concatenate these strings, the **termcap** entry is as shown in Figure F-11.

```
ZA =\E[0;%?%p1%{0}%=%t%{7}%e%p1%{1}%=%t%{3}%e%p1%{2}%=
%t%{5}%e%p1%{3}%=%t%{1}%e%p1%{4}%=%t%{6}%e%p1%{5}%=%t%
{2}%e%p1%{6}%=%t%{4}%e%p1%{7}%=%t%{0}%;%?%p2%t30;%{40}
%+%2d%e40;%{30}%+%2d%;%?%p3%t;5%;%?%p4%t;4%;m
```

Figure F-11 ***Concatenated ZA String for ID231***

In addition to the **ZA** capability, you can use other **termcap** capabilities. **ZG** is the number of character positions on the screen occupied by the attributes of **ZA**. Like the **sg** numeric capability, **ZG** is not required if no extra character positions are needed for display attributes. The value for the **ZG** entry is usually the same value as for the **sg** entry.

The *colornames* File

You can create a **colornames** file if you want to change the default assignment of the names of colors. A **colornames** file is an ASCII file that changes the keywords that you use to write **INFORMIX-4GL** programs. It does not affect the colors produced by your terminal. The format for the **colornames** file follows:

name is the identifier of a color. It cannot be a reserved word and must be unique in the **colornames** file. You cannot assign the same name to more than one number.

number is an integer from 0 to 7.

Usage

Each color name and number must be on a separate line. They should be separated by one or more spaces or tabs.

Unless you redefine them in the **colornames** file to have a different number, the default color-name keywords that are listed in the section titled “Color and Intensity” on page F-29 (and in the next example) retain their meaning, even when you assign another name to that color number.

If you created a **colornames** file to set up the default assignment of names to color numbers, **colornames** would look as follows:

```
WHITE      0
YELLOW     1
MAGENTA    2
RED        3
CYAN       4
GREEN      5
BLUE       6
BLACK      7
```

If you wanted to change CYAN to AQUA and MAGENTA to ORANGE as color names, set **colornames** as follows:

```
AQUA       4
ORANGE     2
```

You could use either CYAN or AQUA in your **INFORMIX-4GL** program and get the same color. Similarly, use of MAGENTA or ORANGE produces the same color.

If you want to change the meaning of the default color names, you can reassign them in **colornames**:

```
RED        2
```

In this case when you use RED in a program, the color you get is the same as has been assigned to MAGENTA. If you have not assigned a name to number 3, you are not able to get the color that RED originally represented.

The **syscolatt** table and the ATTRIBUTE clauses of various **4GL** statements can recognize numeric color codes and non-default names for colors. You can specify these names or numbers in place of the color keywords that are documented in the description of the **upscol** utility in Appendix B.

Note: *You cannot, however, specify numeric codes or non-default names from* ***colornames*** *in the ATTRIBUTES section of a screen form.*

terminfo

If you have set the INFORMIXTERM environment variable to **terminfo**, **INFORMIX-4GL** uses the **terminfo** directory indicated by the TERMINFO environment variable (or **/usr/lib/terminfo** if TERMINFO is not set). **INFORMIX-4GL** uses the information in **terminfo** to draw window borders, define function keys, and display certain intensity attributes.

You may want to modify a file in the **terminfo** directory in the following instances:

- You want to extend function key definitions.
- You want to specify or change the graphics characters used for window borders.
- You want to customize your terminal entry in other ways.

Note: *If you use **terminfo** (instead of **termcap**), you cannot use color attributes with* **INFORMIX-4GL**. *To use color attributes with* **INFORMIX-4GL**, *you must use **termcap**.*

Some terminals cannot support graphics characters. You should read this appendix and the user guide that comes with your terminal to determine whether or not the changes described in this appendix are applicable to your terminal.

To modify a **terminfo** file, you need to be familiar with the following:

- The format of **terminfo** entries
- The **infocmp** program
- The **tic** program

This information is summarized in this appendix; however, you should refer to your operating system documentation for a complete discussion.

Format of a *terminfo* Entry

terminfo is a directory that contains a file for each terminal name that is defined. Each file contains a compiled **terminfo** entry for that terminal. This section describes the general format of **terminfo** entries. For a complete description of **terminfo**, refer to your operating system documentation.

A **terminfo** entry contains a list of names for the terminal, followed by a list of the terminal's capabilities. There are three types of capabilities:

- Boolean capabilities
- Numeric capabilities
- String capabilities

All **terminfo** entries have the following format:

- ESCAPE is specified as a backslash (\ ) followed by the letter E, and CONTROL is specified as a caret (^). Do not use the ESCAPE or CONTROL keys to indicate escape sequences or control characters in a **terminfo** entry.
- Each capability, including the last one in the entry, is followed by a comma (,).

Figure F-12 shows a basic **terminfo** entry for the Wyse 50 terminal:

```
. Entry for Wyse 50:

w5|wy50|wyse50,
  am, cols#80, lines#24, cuul=^K, clear=^Z,
  home=^^, cuf1=^L, cup=\E=%p1%'\s'%+%c%p2%'\s'%+%c,
  bw, ul, bel=^G, cr=\r, cud1=\n, cub1=\b, kpb=\b, kcudl=\n,
  kdub1=\b, nel=\r\n, ind=\n,
  xmc#1, cbt=\EI,
```

Figure F-12 ***Wyse 50*** **terminfo** ***Entry***

Note: *Comment lines begin with a period (.).*

Terminal Names

A **terminfo** entry starts with one or more names for the terminal (each separated by a vertical bar (|)). For example, the **terminfo** entry for the Wyse 50 terminal starts with the following line:

```
w5|wy50|wyse50,
```

The **terminfo** entry can be accessed using any one of these names.

Boolean Capabilities

A Boolean capability is a two- to five-character code that indicates whether or not a terminal has a specific feature. If the Boolean capability is present in the **terminfo** entry, the terminal has that particular feature.

Figure F-13 shows some of the Boolean capabilities for the Wyse 50:

```
     bw,am,

.   bw   backward wrap
.   am   automatic margins
```

Figure F-13 ***Boolean Capabilities for the Wyse 50***

Numeric Capabilities

A numeric capability is a two- to five-character code followed by a pound symbol (#) and a value. Figure F-14 shows the numeric capabilities for the number of columns and the number of lines on a Wyse 50 terminal:

```
     cols#80,lines#24,

.   cols    number of columns in a line
.   lines   number of lines on the screen
```

Figure F-14 ***Numeric Capabilities for the Wyse 50***

String Capabilities

A string capability specifies a sequence that can be used to perform a terminal operation. A string capability is a two- to five-character code followed by an equal sign (=) and a string ending at the next delimiter (,).

Most **terminfo** entries include string capabilities for clearing the screen, cursor movement, arrow keys, underscore, function keys, and so on. Figure F-15 shows many of the string capabilities for the Wyse 50 terminal:

```
      el=\ET,clear=E*,
      cuf1=^L,cuu1=^K,
      smso=\EG4,rmso=\EG0,
      kcuu1=^K,kcud1=^J,kcuf1=^L,kcub1=^H,
      kf0=^A@^M,kf1=^AA^M,kf2=^AB^M,kf3=^AC^M,

.   el=\Et          clear to end of line
.   clear=\E*       clear the screen
.   cufl=^L         non-destructive cursor right
.   cuul=^K         up one line
.
.   smso=\EG4       start stand-out
.   rmso=\EG0       end stand-out
.
.   kcuul=^K        up arrow key
.   kcudl=^J        down arrow key
.   kcufl=^L        right arrow key
.   kcubl=^H        left arrow key
.
.   kf0=^A@^M       function key F1
.   kf1=^AA^M       function key F2
.   kf2=^AB^M       function key F3
.   kf3=^AC^M       function key F4
```

Figure F-15 String Capabilities for the Wyse 50

Extending Function Key Definitions

INFORMIX-4GL recognizes function keys **F1** through **F36**. These keys correspond to the **terminfo** capabilities **kf0** through **kf36**. The **terminfo** entry for these capabilities is the sequence of ASCII characters that your terminal sends when you press the function keys (or any other keys you choose to use as function keys). For the Wyse 50 and Televideo 950 terminals, the first eight function keys send the characters shown in Figure F-16.

Function Key	terminfo Entry
F1	kf0=^A@^M
F2	kf1=^AA^M
F3	kf2=^AB^M
F4	kf3=^AC^M
F5	kf4=^AD^M
F6	kf5=^AE^M
F7	kf6=^AF^M
F8	kf7=^AG^M

Figure F-16 Function Key Entries for the Wyse 50

You can also define keys that correspond to the following capabilities:

- Insert line (**kill**)
- Delete line (**kdll**)
- Next page (**knp**)
- Previous page (**kpp**)

If these keys are defined in your **terminfo** file, **INFORMIX-4GL** uses them. Otherwise, **INFORMIX-4GL** uses CONTROL-J, CONTROL-K, CONTROL-M, and CONTROL-N, respectively.

Note: *You can also use the OPTIONS statement to name other function keys or CONTROL keys for these operations.*

Specifying Characters for Window Borders

INFORMIX-4GL uses characters defined in the **terminfo** files to draw the border of a window. If no characters are defined in this file, **INFORMIX-4GL** uses the hyphen (-) for horizontal lines, the vertical bar (|) for vertical lines, and the plus sign (+) for corners.

You can look at the **terminfo** source file (using **infocmp**) to see if the entry for your terminal includes these definitions. (Look for the **acsc** capability, described later in this section.) If the file for your terminal does not contain border character definitions, or if you want to specify alternative border characters, you or your system administrator can modify the **terminfo** source file. You can refer to your operating system documentation for a complete description of how to decompile terminfo entries using the **infocmp** program.

Perform the following steps to specify border characters in the **terminfo** source file for your terminal:

1. Determine the escape sequences for turning graphics mode on and off. This information is located in the manual that comes with your terminal. For example, on Wyse 50 terminals, the escape sequence for entering graphics mode is ESC H^B and the escape sequence for leaving graphics mode is ESC H^C.

   Note: *Terminals without a graphics mode do not have this escape sequence. The procedure for specifying alternative border characters on a non-graphics terminal is discussed at the end of this section.*

2. Identify the ASCII equivalents for the six graphics characters that **INFORMIX-4GL** requires to draw the border. (The ASCII equivalent of a graphics character is the key you would press in graphics mode to obtain the indicated character.)

   Figure F-17 shows the graphics characters and the ASCII equivalents for a Wyse 50 terminal.

Window Border Position	Graphics Character	ASCII Equivalent
upper left corner	┌	2
lower left corner	└	1
upper right corner	┐	3
lower right corner	┘	5
horizontal	-	z
vertical	|	6

Figure F-17 ***Wyse 50 ASCII Equivalents for Border Graphics Characters***

   Again, this information should be located in the manual that comes with your terminal.

3. Edit the **terminfo** source file for your terminal. (You can decompile it using **infocmp** redirected to a file.)

   Note: *You may want to make a copy of your* ***terminfo*** *directory before you edit files. You can use the TERMINFO environment variable to point to whichever copy of the* ***terminfo*** *directory you want to access.*

Use the following format to enter values for **terminfo** capabilities:

terminfo-capability=value

Enter values for the following **terminfo** capabilities:

smacs The escape sequence for entering graphics mode. In a **terminfo** file, ESCAPE is represented as a backslash (\ ) followed by the letter E; CONTROL is represented as a caret (^). For example, the Wyse 50 escape sequence ESC-H CONTROL-B is represented as \EH^B.

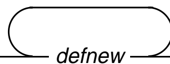
rmacs The escape sequence for leaving graphics mode. For example, the Wyse 50 escape sequence ESC-H CONTROL-C is represented as \EH^C.

acsc The concatenated, paired list of ASCII equivalents for the six graphics characters used to draw the border. You can specify the characters in any order, but you must pair the ASCII equivalents for your terminal with the following system default characters:

Position	System Default Character
upper left corner	l
lower left corner	m
upper right corner	k
lower right corner	j
horizontal lines	q
vertical lines	x

Figure F-18 ***System Default Characters for Border Positions***

Use the following format to specify the **acsc** value:



def is the default character for a particular border character and

new is that terminal's equivalent for the same border character.

For example, on the Wyse 50 terminal, given the ASCII equivalents in Figure F-17 and the system default characters in Figure F-18, the **acsc** capability would be set as shown in Figure F-19.

```
acsc=l2m1k3j5qzx6
```

Figure F-19 ***Wyse 50* acsc *setting***

4. Use **tic** to recompile the modified **terminfo** file. See your operating system documentation for a description of the **tic** program.

   The following example shows the full setting for specifying alternative border characters on the Wyse 50:

```
smacs=\EH^B,          . sets smacs to ESC H CTRL B
rmacs=\EH^C,          . sets rmacs to ESC H CTRL C
acsc=l2m1k3j5qzx6,    . sets acsc to the ASCII equivalents
                      . of graphics characters for upper
                      . left (l), lower left (m), upper right (k),
                      . lower right (j), horizontal (q),
                      . and vertical (x)
```

   If you prefer, you can enter this information in a linear sequence.

```
smacs=\EH^B,rmacs=\EH^C,acsc=l2m1k3j5qzx6,
```

If Your terminfo File Contains xmc#1 Capabilities

The **terminfo** file for some terminals contains xmc#1 capabilities. If xmc#1 is included, 4GL reserves an additional column to the left and right of the window. If you specify a border around the 4GL window, these two columns are in addition to the two additional columns required for the border.

Terminals Without Graphics Capabilities

For terminals without graphics capabilities, you must enter a blank value for the **smacs** and **rmacs** capabilities. For **acsc**, enter the characters that you want **INFORMIX-4GL** to use for the window border.

The following example shows possible values for **smacs**, **rmacs**, and **acsc** in an entry for a terminal without graphics capabilities. In this example, window borders would be drawn using underscores (_) for horizontal lines, vertical bars (|) for vertical lines, periods (.) for the top corners, and vertical bars (|) for the lower corners.

```
smacs=,rmacs=,acsc=l.m|k.j|q_x|,
```

INFORMIX-4GL uses the graphics characters in the **terminfo** file when you specify a window border in an OPEN WINDOW statement.

Color and Intensity

If you use **terminfo**, you cannot use color nor the following intensity attributes in your **INFORMIX-4GL** programs:

BOLD
DIM
INVISIBLE
BLINK

If you specify these attributes in your **INFORMIX-4GL** code, they are ignored.

If the **terminfo** entry for your terminal contains the **ul** and **so** attributes, you can use the UNDERLINE and REVERSE intensity attributes. You can see if your **terminfo** entry includes these capabilities by using the **infocmp** program. Refer to your operating system documentation for information about **info-cmp**.

If you want to use color and intensity in your **INFORMIX-4GL** programs, you must use **termcap** (by setting the INFORMIXTERM environment variable to **termcap**, and by setting the TERMCAP environment variable to **$INFORMIXDIR/etc/termcap**). For more information, see Appendix D.

Appendix

The ASCII Character Set

This Appendix lists the ASCII (American Standard Code for Information Interchange) character set, in ascending order of numeric codes 0 through 127. This ASCII collating sequence is the basis for relational comparisons of strings in **INFORMIX-4GL** and SQL Boolean expressions. These characters appear in the font in which most of this book is typeset.

G

The caret (^) prefix in the first Character column represents the CONTROL key.

<table>
<tr><th>Code,</th><th>Character</th><th>Code,</th><th>Character</th><th>Code,</th><th>Character</th></tr>
<tr><td>0</td><td>^@</td><td>43</td><td>+</td><td>86</td><td>V</td></tr>
<tr><td>1</td><td>^A</td><td>44</td><td>,</td><td>87</td><td>W</td></tr>
<tr><td>2</td><td>^B</td><td>45</td><td>-</td><td>88</td><td>X</td></tr>
<tr><td>3</td><td>^C</td><td>46</td><td>.</td><td>89</td><td>Y</td></tr>
<tr><td>4</td><td>^D</td><td>47</td><td>/</td><td>90</td><td>Z</td></tr>
<tr><td>5</td><td>^E</td><td>48</td><td>0</td><td>91</td><td>[</td></tr>
<tr><td>6</td><td>^F</td><td>49</td><td>1</td><td>92</td><td>\</td></tr>
<tr><td>7</td><td>^G</td><td>50</td><td>2</td><td>93</td><td>]</td></tr>
<tr><td>8</td><td>^H</td><td>51</td><td>3</td><td>94</td><td>^</td></tr>
<tr><td>9</td><td>^I</td><td>52</td><td>4</td><td>95</td><td>_</td></tr>
<tr><td>10</td><td>^J</td><td>53</td><td>5</td><td>96</td><td>`</td></tr>
<tr><td>11</td><td>^K</td><td>54</td><td>6</td><td>97</td><td>a</td></tr>
<tr><td>12</td><td>^L</td><td>55</td><td>7</td><td>98</td><td>b</td></tr>
<tr><td>13</td><td>^M</td><td>56</td><td>8</td><td>99</td><td>c</td></tr>
<tr><td>14</td><td>^N</td><td>57</td><td>9</td><td>100</td><td>d</td></tr>
<tr><td>15</td><td>^O</td><td>58</td><td>:</td><td>101</td><td>e</td></tr>
<tr><td>16</td><td>^P</td><td>59</td><td>;</td><td>102</td><td>f</td></tr>
<tr><td>17</td><td>^Q</td><td>60</td><td><</td><td>103</td><td>g</td></tr>
<tr><td>18</td><td>^R</td><td>61</td><td>=</td><td>104</td><td>h</td></tr>
<tr><td>19</td><td>^S</td><td>62</td><td>></td><td>105</td><td>i</td></tr>
<tr><td>20</td><td>^T</td><td>63</td><td>?</td><td>106</td><td>j</td></tr>
<tr><td>21</td><td>^U</td><td>64</td><td>@</td><td>107</td><td>k</td></tr>
<tr><td>22</td><td>^V</td><td>65</td><td>A</td><td>108</td><td>l</td></tr>
<tr><td>23</td><td>^W</td><td>66</td><td>B</td><td>109</td><td>m</td></tr>
<tr><td>24</td><td>^X</td><td>67</td><td>C</td><td>110</td><td>n</td></tr>
<tr><td>25</td><td>^Y</td><td>68</td><td>D</td><td>111</td><td>o</td></tr>
<tr><td>26</td><td>^Z</td><td>69</td><td>E</td><td>112</td><td>p</td></tr>
<tr><td>27</td><td>esc</td><td>70</td><td>F</td><td>113</td><td>q</td></tr>
<tr><td>28</td><td>^\</td><td>71</td><td>G</td><td>114</td><td>r</td></tr>
<tr><td>29</td><td>^]</td><td>72</td><td>H</td><td>115</td><td>s</td></tr>
<tr><td>30</td><td>^^</td><td>73</td><td>I</td><td>116</td><td>t</td></tr>
<tr><td>31</td><td>^_</td><td>74</td><td>J</td><td>117</td><td>u</td></tr>
<tr><td>32</td><td></td><td>75</td><td>K</td><td>118</td><td>v</td></tr>
<tr><td>33</td><td>!</td><td>76</td><td>L</td><td>119</td><td>w</td></tr>
<tr><td>34</td><td>"</td><td>77</td><td>M</td><td>120</td><td>x</td></tr>
<tr><td>35</td><td>#</td><td>78</td><td>N</td><td>121</td><td>y</td></tr>
<tr><td>36</td><td>$</td><td>79</td><td>O</td><td>122</td><td>z</td></tr>
<tr><td>37</td><td>%</td><td>80</td><td>P</td><td>123</td><td>{</td></tr>
<tr><td>38</td><td>&</td><td>81</td><td>Q</td><td>124</td><td>|</td></tr>
<tr><td>39</td><td>'</td><td>82</td><td>R</td><td>125</td><td>}</td></tr>
<tr><td>40</td><td>(</td><td>83</td><td>S</td><td>126</td><td>~</td></tr>
<tr><td>41</td><td>)</td><td>84</td><td>T</td><td>127</td><td>del</td></tr>
<tr><td>42</td><td>*</td><td>85</td><td>U</td><td></td><td></td></tr>
</table>

Appendix

Reserved Words

This appendix lists keywords that you should not use as programmer-defined identifiers in an INFORMIX-4GL program. If you do, the program may fail with a compilation or run-time error, or may produce unexpected results. (If you receive a system error message that seems to be unrelated to the statement that produced the error, you should review this appendix to see if the error may have been caused by a reserved word used as an identifier.)

In general, you cannot use as an identifier the name of a built-in constant or variable, nor the name of an operator that can begin an expression. Chapter 4 describes restricted functionality that results if you defined a function or report with the same name as a built-in 4GL function (page 4-6) or a 4GL operator (page 4-10).

You are permitted to declare most other keywords of 4GL as identifiers, but you may not be able to reference the identifier in contexts where the keyword makes sense. (For example, if you open a 4GL window named SCREEN, you will not be able to reference it in statements like CURRENT WINDOW where SCREEN specifies the 4GL screen.) Your code is likely to be difficult to read and to maintain if you use keywords as identifiers.

See the *Informix Guide to SQL: Tutorial* for information about the use of reserved words as SQL identifiers.

In addition to the following words, do not declare the names of operating system calls, nor of C or C++ language keywords as identifiers in your 4GL programs.

Do not use any of the following words as 4GL identifiers:

ASCII
AVG
COLUMN
CONSTANT
COUNT
COPY
CURRENT
DATE
DATETIME
DAY
EXTEND
FALSE
FIELD_TOUCHED
GET_FLDBUF
INFIELD
INTERVAL
INT_FLAG
LENGTH
LINENO
MAX
MDY
MIN
MONTH
NEW
NOT
NOTFOUND
NOW
NULL
PAGENO
PERCENT
QUIT_FLAG
SQLCA
STATUS
SUM
TIME
TRUE
TODAY
WEEKDAY
YEAR

Developing Applications for International Markets

The emerging global economy is creating opportunities and challenges for software application developers. Increasingly, applications are targeted toward international environments (such as multi-national corporations) or are intended for customers in different countries. Meeting the needs of these *global users* requires that an application include international features such as translatable user messages, forms, menus and reports, as well as date, time, money and currency formats that can be easily changed to fit local cultural standards. Application developers now are faced with the difficult task of *internationalization*, or making their applications world-ready.

This appendix describes the internationalization features provided with INFORMIX-4GL and shows how you can develop **4GL** applications that are world-ready and easy to localize.

What Is Internationalization?

Internationalization is the process of making software applications easily adaptable to different cultural and language environments. Internationalization features include support for adaptable date, time and money formats, and the use of environment variables to switch the run-time environment from one language to another.

A fully internationalized application can run in different cultural environments with minimal adjustments; in some instances, you can simply exchange language-specific files and set up the proper operating environment. Internationalization removes the need to recompile source code for a specific language or cultural environment.

An internationalized application should be ***8-bit clean***. If a software program or operating system allows the high order bit of a character code to take on a value of 1, it is referred to as 8-bit clean. 4GL products are 8-bit clean, allowing the use of Extended ASCII character sets such as IBM PC code pages, or ISO 8859 character sets. An extension of the technology allows usage of double-byte and multi-byte character sets with the Asian Language Support versions of 4GL.

What Is Localization?

Localization is the process of translating and adapting an internationalized product to specific language and cultural environments. Localization usually involves setting the appropriate date, time, and money formats for the intended country as well as creating a translation of the product user interface (including help and error messages, prompts, menus, and forms and reports).

When internationalization is built into the application from the start, the localization effort will be significantly easier.

Developing Applications for the Global Market

Increasingly, 4GL applications are being deployed in the global market place. To develop these internationalized applications, developers create their own menus, forms, reports, and customized help facilities. These elements of the application will potentially have to be translated into foreign languages.

Requirements for International Application Development

Developing an application that is fully adapted to a country requires the following:

- The targeted hardware platform and operating system need to support the desired language/country combination.

  Special versions of the operating system environment on both the client platform and the server platform may be required to support the entry, manipulation, and display of non-English data.
- The Informix product(s) need to support the language.

  Informix products are 8-bit clean and allow entry, manipulation, and display of most European language data. A special version of Informix products may be required, however, to support Asian or Arabic languages.
- A localized version of the error messages generated by the application development language and the database server should be available for the requested languages to provide a localized run-time environment.
- All parts of the user interface created by the application developer, such as menus, forms, error messages, and help should be translated into the target language.

As noted earlier, you can reduce localization cost and effort if the application is designed with international requirements in mind. The following internationalization guidelines will help you reduce the cost of your localization efforts.

Internationalizing Applications

To make a 4GL application world-ready, keep the following guidelines in mind.

- Make sure the targeted hardware and operating system environment, as well as the version of the Informix products you are using in your applications, support the desired language and culture.
- Find out if a language translation (localization) is available for the run-time environment of the application development tool and database server you are using.

  For example, the 4GL development and run-time environments (and the INFORMIX-OnLine administrator's environment) are usually translated into several languages including French, German, Spanish, and Italian.

- Use the concept of 4GL libraries where possible. This centralizes common code and makes changes and maintenance easier when developing for international markets.
- Use variables instead of literal strings where possible in your **4GL** application. These variables can then be initialized as appropriate for the specific language environment. Three possible approaches follow:
    - You can obtain the information for initializing the variables from a database table using SQL queries. This is commonly referred to as a *table-driven approach.*
    - You can use a custom C function to retrieve the value of the variables from a text file. This is referred to as a *file-based approach.*
    - You can use environment variables to initialize 4GL variables with language-specific values.

  Your specific application deployment and target environment will determine the best approach. Each of the methods available for "externalizing" language-specific elements of the source code has specific advantages, some of which are summarized at the end of this appendix.
- Make sure that all user messages, reports, and help facilities that were developed with Informix tools for the application are either table driven or are controlled by text files or environment variables that are easy to modify. Keep in mind that forms need to be recompiled after they have been modified or adapted to a foreign language.
- Avoid embedding or *hard coding* any messages, prompts, or elements of the user interface into the source code of the program. Ideally, all user interface elements can be switched dynamically by *referencing* a different set of translated files.
- Make important application parameters such as holidays, bank years, and formulas table-driven or file-based.
- Consider different keyboard layouts: A character easily accessible on an US keyboard (such as "/") might require several key strokes in a foreign country.
- Allow space for the expansion of user message strings.

  Brief English strings such as *Pop-Up* can double in size as a result of translation. On average, you can expect a 30% increase in the size of messages.
- Avoid fragmentation of messages or potentially ambiguous key or command words. Always place comments around any string that pertains to the user interface to facilitate localization.

- Use custom error messages and help files.

  The **mkmessage** utility allows you to create custom help files as well as a localized version of the **4GL** run-time message file. (This is the **4glusr.msg** file in the **$INFORMIXDIR/msg** directory.)

- Use environment variables to specify the locale of the 4GL application. Several environment variables affect your **4GL** application. For example:
  - DBLANG points to the message directory for Informix error messages and month and day names.
  - DBFORM points to the menu form directory for customized **4GL** system menus.
  - DBFORMAT and DBMONEY defines numeric and monetary formats.
  - DBDATE defines date and time display.
  - All the NLS environment variables.

  (For complete information on using NLS, see Appendix E, "Native Language Support Within INFORMIX-4GL.")

- Do not assume that the users of your application will speak English or will accept any pre-set business rules or formats.

- Make use of subdirectories where possible to store language-sensitive files, so they can be easily switched to create a new run-time environment.

  Alternatively, you can develop your own language variable scheme. For example, the following three letters identify a unique subdirectory that contains the compiled form specification files appropriate for a particular language:

  eng = English
  fre = French
  ita = Italian
  spa = Spanish

  Using this approach, the names of form files can be composed and referenced by reading the value of an environment variable that specifies the language subdirectory:

```
LET file001 = FGL_GETENV("LANGUAGE"), "/", "form001.frm"
# Evaluates to "eng/form001.frm" if LANGUAGE is "eng"
# Program reads the eng directory for copy of form001
#
# Evaluates to "ger/form001.frm" if LANGUAGE is "ger"
# Program reads the ger directory for copy of form001
#
OPEN WINDOW w1 AT 5, 5  WITH FORM file001
```

Preparing a Translation Checklist

You may want to create a checklist of user interface elements in your application that should be *externalized* from the source code, and therefore from the compiled portion of the program into text files. These text files can then serve as "resource" files to the **4GL** application and can be modified even after the program is compiled.

Elements to externalize include the following:

- Menus
- Messages
- Prompts/Dialog responses
- Command keys
- Help (.msg) text
- Form (.per) files
  - Field labels
  - Format attributes (dates, times)
  - Comments attribute
  - Include attribute
- Report names

File-Based Internationalization

The following example shows how a record can be created to hold custom error messages, and how the record can be initialized by reading an external message file. This is just one example of a possible internationalization approach for this situation and it assumes that you will use a C code function to access the file. As mentioned previously, the messages record could also be initialized by fetching the data from a database table.

If you provide custom C code functions for file access, make sure the C code is 8-bit clean and does not strip out characters in the extended 8-bit character range where the most significant bit is set to 1.

First create a record for all messages:

```
STR record
    mssg1 char(40),# Enter Name
    mssg2 char(80),# Validate input by pressing "Accept"
    . . .
end record
```

Next, use a C function to initialize the variables from a file:

- For a **C Compiler Version** application, link the C function with the application.
- For an **RDS Version** application, make sure that the custom runner includes the function.

```
    let file_01 = "msgs_01.", FGL_GETENV ("LANGUAGE")
    let STR.message_01 = C_get_message(file_01, 0001)
    let STR.message_02 = C_get_message(file_01, 0002)
```

Two parallel initialization files follow; the first is in English, and the second is in German.

```
# Contents of text file "msgs_01.eng"

# This file contains messages and prompts for module A.
# Note to Translator! Don't exceed buffer length specified in brackets!

0001: "Enter name", <40>
0002: "Validate input by pressing "Accept"", <80>
0003: "File not found", <40>
0004: "A serious error has occurred, please contact Technical Support", <80>
0005: "", <80>
0006: "", <80>
. . .
```

```
# Contents of text file "msgs_01.ger" after translation
# (Stored in TRNSLATE/German directory)

# This file contains messages and prompts for module A.
# Note to Translator! Don't exceed buffer length specified in brackets!
# Translated for German customer Schulz on October 31, 1993

0001: "Namen eingeben", <40>
0002: "Eingabe bitte mit "Accept" bestätigen", <80>
0003: "Datei nicht gefunden", <40>
0004: "Schwerwiegender Fehler! Bitte Kundendienst anrufen!, <80>
0005: "", <80>
0006: "", <80>
. . .
```

Setting the environment variable LANGUAGE to either "eng" or "ger" changes the messages from English to German without requiring you to recompile the **4GL** application source code.

Table-Based Internationalization

As noted earlier, localization information can also be stored in database tables. This information can be used when you initialize the application or as the application runs to change the value of variables defining messages, prompts, menus, and other language or culturally sensitive data.

An advantage of the table-based approach is that it is highly portable between systems. It is easier to implement than a file-based approach since it does not require custom C functions.

The following is an example of a table to store menu options.

```
CREATE TABLE menu_elements(
    msg_languageCHAR(3),# language ID code
    msg_numberSMALLINT,# message number
    msg_textCHAR(80),# message text
    msg_maxlenSMALLINT# maximum length of message
    )

CREATE UNIQUE INDEX ix_errormesg ON menu_elements(msg_language, msg_number)
```

Example data:

```
ENG 150 Cold Beer
FRE 150 Bière froide
GER 150 Kaltes Bier
SPA 150 Cerveza fría
ENG 151 Iced Tea
...
```

A global variable with the application's language code (corresponding to the value in the **msg_language** column) could be set in the program upon start-up. Then, every time a message is needed, a function could be called that uses the language code and message code to identify the appropriate message string. For example:

```
### program startup
LET g_language = get_language()# returns 3 letter code corresponding
    # to value in msg_language column
### main ring menu
LET p_str = get_message(150, g_language)
MENU p_str
```

Forms, Reports, Message Files, and Help Files

Since you cannot use **4GL** variables to specify text (for example, field labels) within a form file, you should use one of two approaches to specify the appropriate form:

- Create a set of modified forms for each language you plan to support.

  You can then store each set of forms in a language subdirectory which can then be selected by the application. You can use environment variables to specify the appropriate set of forms, or provide a menu and allow the user to make the selection.
- Remove all literal text (this includes field labels) from your form specifications and use **4GL** DISPLAY statements to display this information.

  The DISPLAY statements can reference variables that are initialized to contain different values depending on the language wanted.

You can handle reports (which are **4GL** programs) same way you internationalize the rest of your **4GL** source code. Make sure to leave sufficient space in your headers and titles to accommodate the potentially longer text of a foreign language translation.

Finally, you can use the **mkmessage** utility to customize the error message files delivered in source code format and to create help files.

Localizing Your Applications

As noted previously, localization refers to the actual process of adapting the application to the cultural environment of the end users. This process often involves translation of user interface and user documentation and can be quite time consuming and costly. Here are some guidelines to follow.

- Consult the native operating system internationalization guide.

  Most platforms provide documentation on internationalization. This material may help you to determine which date/time and money formats are appropriate for the target language/culture.

- Keep a glossary of all strings and keywords in a database or text file.

  This glossary of terms and strings will make it easier to see which messages are duplicated throughout the source code. The glossary will also increase consistency of terms and language of the user interface throughout the application. Once the glossary is created for one language, it can be used for product updates and additional localizations.

- Create a mechanism that allows a glossary to drive the definition of the user interface.

  This can be particularly useful if you expect to localize the application often. A translator can edit the glossary without having to understand the source code of the application. Your tool can then create the user interface from the translated glossary, and the translator can focus on making cosmetic enhancements to the translation (such as positioning the messages appropriately) and correcting minor errors.

- Consider retaining a professional translator for some or all of this process.

  A faulty translation is very costly. You can spend a great deal of time and money correcting errors in your localized product. And, if you do not correct the problems, your users will be dissatisfied with your application.

Internationalization Methodology Overview

The following table lists the different elements of an application and indicates how each can be internationalized. This overview, while not comprehensive, illustrates how to approach a project of this nature. In several instances, table-driven, file-based, and environment-variable based approaches are contrasted.

Application Element	Method Available	Comments
Menus, Prompts, Command Keys, Report titles, names, and other application variables	Table-driven	Assumes availability of database tables. Requires hard coding of defaults in case database tables cannot be accessed. Portable approach, but requires database access for translation. Can cause performance degradation in client-server environments because of the additional number of queries required. To improve performance issues in client-server environments, initialize the most common language variables at application start-up.
	File-based	Assumes customized C code function to access files. Text files can be easily modified to create additional translations. Potential limitation of file naming schemes or operating environment. National language characters can have a different encoding on client and server platforms.This approach provides a solution to the encoding problem across multiple platforms. May yield better performance than the table-driven method since fewer SQL queries are required overall.
	Environment variable-based	Assumes sufficient space in environment to store all information. Easy to set up and maintain on a single platform, but can lead to more complex portability issues between platforms. The main advantage of this method is that there is no need for a customized C code function to access text files and no additional SQL queries to obtain information from tables.
Help and Error messages	Table-driven	Assumes availability of tables. Often needs hard coded defaults in case data tables cannot be accessed.
	File-based	Use the **mkmessage** utility to create help files and to modify error message files (if source code for error message files is delivered with the product).
Forms	Translation	Forms need to be translated and recompiled to adapt to foreign languages and different cultural environments. Store in separate subdirectories and select using user menus or a language naming scheme. Alternate is to remove all literal text from each form specification and to use DISPLAY statements to display this information on the form.
Date, Time, Money Formats	Informix environment variables	Informix Environment variables DBMONEY and DBDATE allow to adapt the display of date, time and money formats to cultural conventions.

Figure I-1 Internationalization Methods

Application Element	Method Available	Comments
Informix error messages	Informix translation	Informix provides error message translation for a variety of languages.You can use the DBLANG environment variable to point to a message directory containing translated messages. Contact your local Informix sales office for a list of available language translations.
	Custom error message files	If no Informix translation of the error messages is available, and if the source code of error message files are delivered with the product, you can localize the message source file(s) using the **mkmessage** utility.

Figure I-1 Internationalization Methods

Glossary

4GL function A 4GL program block defined with the FUNCTION statement. The function header follows the FUNCTION keyword and defines the name and formal argument list for the function. The function body (all statements between the function header and the END FUNCTION keywords) defines the actions of the function. The function header and the function body together are often called the "function definition." To return values, use the RETURN statement within the function body. Frequently, a 4GL function is simply referred to as a "function." See also *argument, function, programmer-defined function, program block, return value.*

4GL **screen** 1) In the INFORMIX-4GL Interactive Debugger, the 4GL screen is where the Debugger displays the 4GL application.

2) When running a 4GL application, the 4GL screen is the display area of the screen; this area displays the application's forms, 4GL windows, and text. See also *4GL window, screen.*

4GL **window** A rectangular region in the 4GL screen, possibly one of many, managed by a 4GL application. The default 4GL window, is the 4GL screen. The OPEN WINDOW statement creates a new 4GL window. 4GL manages its windows with a stack. Each window is pushed onto this stack when it is opened. A 4GL program performs its input and output in the current window. See also *4GL screen, current, popup window, reserved lines, screen, stack.*

abnormal termination The termination of the 4GL application through any mechanism other than exiting the MAIN program block at the END MAIN keywords or with a RETURN statement. A run-time error, pressing an Interrupt or Quit key, or executing the EXIT PROGRAM statement result in an abnormal termination. In the INFORMIX-4GL Interactive Debugger, you can inspect the application state after an abnormal termination. You cannot, however, resume execution. See also *debug, exception handling, MAIN program block, normal termination, program execution.*

Accept key The logical key that the user can press within a 4GL application to indicate acceptance of the entered data or query criteria. Pressing it requests normal completion of a user interaction statement. By default, the physical key for Accept is ESCAPE. See also *data entry, Interrupt key, logical key, query criteria, Quit key, user interaction statement.*

activation key A logical key that the developer defines to provide the user with some programmer-defined feature. The developer can define an activation key in the ON KEY clause of the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or PROMPT statement; or in the KEY clause of the MENU statement. When the user presses the activation key, 4GL executes the control block associated with that key. See also *control block, key, logical key, menu option.*

active form A screen form for which the current user interaction statement is executing. An application can have several active forms if it has several 4GL windows, each continuing a screen form with a user interaction statement executing. With multiple active forms, only one form is current. See also *4GL window, current, screen form, user interaction statement.*

active function A 4GL program block (including MAIN, a function, or a report) that has started execution but not completed execution. The active functions consist of all functions on the call stack—that is, of the current function and all functions that are waiting for a function call to return. The MAIN program block is always active for the program session. You can inspect active functions within the INFORMIX-4GL Interactive Debugger. See also *active variable, abnormal termination, call stack, debug, normal termination, program block, program execution.*

active set The collection of database rows satisfying a query associated with a database cursor. An active set is stored in memory at run time. It contains only a copy of the rows that match the query criteria. See also *cursor, query, row.*

active variable A 4GL variable for which storage exists. The active variables consist of the local variables of all active functions, the module variables of all modules containing the active functions, and all global variables. You can evaluate

or assign values to active variables within the INFORMIX-4GL Interactive Debugger. See also *active function*, *abnormal termination*, *global variable*, *local variable*, *module variable*, *normal termination*, *program execution*, *scope*.

active window The window that contains the current keyboard focus. See also *keyboard focus*.

actual argument In a function call, the value being passed by the calling routine as an argument to the programmer-defined function. This value must be of a compatible data type with the corresponding formal argument in the function definition. There are two methods for an actual argument to be passed to a function: pass-by-reference and pass-by-value. See also *argument*, *calling routine*, *data type conversion*, *formal argument*, *function call*, *pass-by-reference*, *pass-by-value*, *programmer-defined function*.

aggregate function 1) A function built into the SQL language that returns a single value based on the values of a column in several rows of a table. These functions are called SQL aggregate functions. Examples of SQL aggregate functions are SUM( ), COUNT( ), MIN( ), and MAX( ). These are valid only within SQL statements. See also *column*, *row*, *SQL*, *table*.

2) A function built into the 4GL language that returns a single value based on the values of input records. These functions are called report aggregate functions. Examples of report aggregate functions are SUM( ), GROUP SUM( ), PERCENT(*), MIN( ), and MAX( ). Report aggregate functions are valid only within a REPORT program block. See also *built-in function*, *input record*, *program block*, *report*.

alias In the database, an alias is a synonym for a table name. It immediately follows the name of the table in an SQL statement. It can be used wherever the name of a table can be used. Aliases are often used to create short, readable names for long or external table names. See also *table*.

ANSI-compliant A database that conforms to certain ANSI (American National Standards Institute) performance standards. Informix databases can be created either as ANSI-compliant or as not ANSI-compliant. An ANSI-compliant database enforces ANSI requirements such as implicit transactions, required owner-naming, and no buffered logging. The term MODE ANSI is sometimes used to refer to an ANSI-compliant database. See also *database*, *implicit transaction*.

application development tool Software, such as INFORMIX-SQL, INFORMIX-4GL, and **INFORMIX-ESQL**, which a developer can use to create and to maintain a database. Such software allows a user to send instructions and data to and to receive informa-

tion from a database server. An application development tool is sometimes referred to as the "front end" and the database engine as the "back end." See also *database engine*.

application program 1) A computer program developed and implemented for dealing with some business activity. (A computer program is a group of instructions that cause a computer to do a sequence of operations.) An application program is synthesized from some combination of application development tools. See also *application development tool*, *database*, *developer*, *user*.

2) In 4GL, an application program is the 4GL program, with one MAIN program block, its supporting 4GL source modules, its form specification files, and its help message file. See also *form specification file*, *help file*, *MAIN program block*, *source module*.

application program interface (API) A rigorous definition of the method by which a program can access the services provided by another program. Developers of an API often provide libraries of callable functions that implement the API. Examples include: the 4GL API enables a C program to call a 4GL routine; Motif enables a C program to call X Windows; **INFORMIX-ESQL**, which is an SQL API, enables a C program to access a database. There can be many different APIs that provide access to the same set of services, though possibly at different points of entry. In some cases, the same API can be used to access different services. (Example: NetBIOS is a network API that is protocol-independent and often used to access a variety of different protocols such as OSI, TCP/IP, and so on.) See also *application development tool*.

argument A value passed from a calling routine to a function. In the calling routine, the value passed is called an "actual argument." Within the function definition, the name of the argument is called a "formal argument." When the function is called, the value of the actual argument is assigned to the corresponding formal argument variable. See also *actual argument*, *calling routine*, *formal argument*, *pass-by-reference*, *pass-by-value*, *programmer-defined function*, *report*.

arithmetic operators Operators that perform arithmetic operations on operands of number (and some time) data types. The following are 4GL binary arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), exponentiation (**), and modulus (MOD). 4GL unary arithmetic operators are: unary minus (-) and unary plus (+). The following precedence is for arithmetic operators: (highest) unary minus and plus; (next highest) exponentiation, modulus; (next highest) multiplication, division; (lowest) addition, subtraction. Arithmetic operators yield a numeric result. See also *associativity*, *binary operator*, *operand*, *operator*, *precedence*, *unary operator*.

array 1) A data structure having a fixed number of components. Each component is called an element. All elements in an array have the same data type. See also *array element*, *screen array.*

2) In uppercase, ARRAY is the keyword for defining a program array in 4GL. The ARRAY data type is a structured data type of up to three dimensions. It cannot have another array as an element. An array element is accessed by listing the array name followed by a subscript. See also *program array, structured data type*, *subscript.*

array element A component of a program array. An element can be of any 4GL data type, except ARRAY. To reference the position of any element within an array, use a subscript (sometimes called an "array index"). See also *array, program array, subscript.*

ASCII 1) Acronym for American Standards Code for Information Interchange. Often used to describe the ordered set of internal codes which a computer uses to represent characters. This set includes both printable and non-printable characters. Appendix G contains a list of the ASCII character set. See also *control character, escape character, printable character.*

2) An ASCII file is one containing ASCII characters as opposed to binary information. An ASCII file is readable in a text editor. 4GL source modules and form specification files are examples of ASCII files. See also *form specification file*, *source module*, *text editor.*

3) In NLS, ASCII can refer to either the ASCII character set, which is a set of 128 characters and their numeric representations, or ASCII collation, which is the ordering of characters in the ASCII character set by their numeric values. See also *Native Language Support.*

assign To store a value in a variable. The 4GL assignment operator is the LET statement. 4GL evaluates the expression on the right-hand side of the "=" and assigns it to the variable listed on the left-hand side. You can also assign values to a variable with the following 4GL statements: CALL...RETURNING, FOREACH...INTO, INITIALIZE, INPUT, INPUT ARRAY, and PROMPT; and with the INTO clause of the SQL SELECT statement. In evaluation of expressions, assignment has the lowest precedence. See also *expression*, *operator, precedence*, *variable.*

associativity The principle that determines the order in which operands at the same level of precedence in an expression are evaluated. For example, to evaluate the expression **a** - **b** + **c**, 4GL first evaluates **a** - **b** and then adds **c** to the result because binary arithmetic operators associate to the left. See also *binary operator*, *expression*, *precedence.*

asterisk notation The syntax ".*" appended to the name of a table (*table.**) or a program record (*record.**). This syntax expands to the names of all columns in a table or of all members in a record. See also *program record*, *table*.

attribute 1) A characteristic or aspect of some entity which the developer can set. 4GL provides attributes for form fields (field attributes), screen forms (form attributes), database columns (column attributes), and for output text (display attributes). Field attributes are set on a field-by-field basis in the form specification file. Form attributes are set with the ATTRIBUTES clause of the DISPLAY FORM statement and of the 4GL user interaction statements. Display attributes can also be set with the ATTRIBUTES clause. Column attributes are set on a column-by-column basis with the **upscol** utility. See also *column*, *field*, *form*, *form specification file*, *user interaction statement*.

2) In some database terminologies, a term used for a "column." See also *column*.

background process In a multi-processing environment, a process that is not performing input or output. It can continue to run without needing access to a window or the screen. See also *foreground process*, *process*, *screen*.

batch A mode of execution in which a program runs without input from a user. 4GL programs that do not use user interactive statements are batch programs. If a batch program produces output, it should direct the output to a file or the printer, not the screen. Often reports are run in a batch mode. See also *interactive*, *program execution*, *report*, *user interaction statement*.

binary operator An operator that requires two operands. The binary operator appears between the two operands. In 4GL, examples of binary operators include addition (+), multiplication (*), and logical AND. 4GL associates most binary operators from left-to-right. See also *arithmetic operators*, *associativity*, *Boolean operators*, *operand*, *operator*, *precedence*, *relational operators*, *unary operator*.

binding A one-to-one correspondence between entities in two domains. The association between an identifier and its resource (a location in memory) is called a binding. In 4GL, the correspondence between form fields and program variables during data entry is also called a binding. Several 4GL user interaction statements include a binding clause that lists the names that accept user input from a form field and program variables and their corresponding form fields (or database columns). These statements include CONSTRUCT, INPUT, INPUT ARRAY, and PROMPT. See also *data entry*, *identifier*, *program array*, *program record*, *screen array*, *screen record*, *user interaction statement*.

blank space The character with the value of ASCII 32. A string of blank spaces is not the same as a NULL string (which has nothing in it). 4GL pads string values with blank spaces up to the size of the CHAR or VARCHAR variable. Spaces are also used to separate elements on a form. Also referred to simply as "blanks." See also *ASCII*, *clipped*, *null value*, *printable character*, *string*.

blob An acronym for Binary Large Object. In 4GL, blob data types can hold values that occupy up to 2^{31} bytes. These data types include TEXT (character data) and BYTE (binary data). See also *byte*, *data type*, *text*.

Boolean 4GL includes two Boolean constants: FALSE (= 0) and TRUE (=1). If an operand evaluates to NULL, Boolean operators can yield an "unknown" result that 4GL treats as FALSE, in some contexts. Because 4GL does not have a Boolean data type, Boolean values should be stored in the integer data types. See also *Boolean operators*, *constant*, *integer*, *relational operators*.

Boolean operator An operator that returns a Boolean value. In some contexts, 4GL interprets any operand that is NULL as having a value of FALSE and any non-zero operand as TRUE. Boolean expressions can also include relational operators. See also *associativity*, *binary operator*, *Boolean*, *operand*, *operator*, *precedence*, *relational operators*, *unary operator*.

built-in function A function which is part of the 4GL language and can therefore be called from a calling routine without needing to be defined by the developer. Function calls for built-in functions have the same syntax as those for programmer-defined functions. Examples of built-in functions are: ARG_VAL( ), ARR_CURR( ), FGL_KEYVAL( ), and SCR_LINE( ). See also *4GL function*, *aggregate function*, *built-in operator*, *calling routine*, *function call*, *programmer-defined function*.

built-in operator An operator which is part of the 4GL language. Built-in operators are keywords that perform special tasks. They differ from built-in functions in that they cannot be invoked with the CALL statement and they cannot be called from a C function. Examples of built-in operators are: ASCII, CURRENT, DATE, and TODAY. See also *built-in function*, *keyword*, *operator*.

byte 1) A unit of storage, corresponding to one character. A kilobyte is 1,024 bytes. A megabyte is 1,048,576 bytes. See also *character*.

2) In uppercase letters, BYTE is the **4GL** and SQL data type that can store up to 2^{31} bytes of binary data. See also *blob*.

C Compiler Version The version of 4GL that precompiles 4GL code into INFORMIX-ESQL/C code and then translates the **ESQL/C** into object code, executable directly from the command line. See also *compile, Rapid Development System*.

call stack A stack used by 4GL at run time to keep track of active functions. An active function is one that has been called but that has not yet returned. Each time the program calls a function, the function's state is pushed onto the call stack. When a function exits, the state is popped off the call stack. The MAIN program block is always at the bottom of this stack. You can examine the call stack within the INFORMIX-4GL Interactive Debugger. See also *active function, MAIN program block, stack.*

call-by-reference See *pass-by-reference.*

call-by-value See *pass-by-value.*

calling routine The program block that calls a function (or report). In 4GL, the calling routine can be either a MAIN, a FUNCTION, or a REPORT program block. This call can be explicit, by means of the CALL statement, or implicit, by embedding the function name within an expression. The calling routine can pass in values through the function arguments and can receive return values (if they are defined within the function). See also *argument, expression, function call, program block, return value.*

case sensitivity The ability to distinguish between uppercase and lowercase letters. The 4GL language is not case-sensitive. The variables **a** and **A** refer to the same address in memory. Certain command-line syntax (command names and options) are case-sensitive. See also *identifier, keyword, naming conventions.*

category In NLS, category refers to each feature of the locale, pertaining to one aspect of the language and formatting environment. Standard NLS categories include collation, character set, monetary formatting, numeric formatting, and date/time formatting. Date/time formatting is not a supported NLS category in 4GL. See also *Native Language Support.*

character 1) Any letter, digit, symbol, or control sequence that can be represented by the ASCII character set. See also *ASCII, blank space, control character, escape character, printable character.*

2) The character data types are CHAR and VARCHAR (and in some contexts, a TEXT variable). See also *blob, data type, string, string operators, subscript.*

3) What a single keystroke, control character, or escape sequence produces that the program, operating system, or output device treats as a single unit. See also *activation key, logical key, operating system.*

character set A set of valid characters, each of which corresponds to an integer value from 0 to 255 (8-bit) or 0 to 127 (7-bit). In NLS, a character set is specified by way of the name of a character set file. A character set file contains all

of the characters and their corresponding numeric values. The NLS character sets are provided to meet the needs of European countries. They extend the ASCII character set used for English, which consists of 128 characters (there are 128 possible combinations of 7 bits of data), to one of several possible sets of 256 characters (based on 8 bits per character). The most prevalent of the 256-character sets are the ISO 8859-1 and 8859-2 standards. See also *ASCII, Native Language Support.*

clipped The CLIPPED keyword is a built-in operator that removes trailing blank spaces from strings. It is often used in DISPLAY and PRINT statements. See also *blank space, built-in operator, string operators.*

close To cease to use an open entity. In programming, closing something releases control of it and deallocates any resources that it used. For example, when you close a database cursor, you release any memory or disk space that was used to hold the active set for that cursor. When you close a file, you tell the system that you no longer require the file and others can use it. When you close a form, you release any memory or disk used to store it. When you close a 4GL window, you deallocate memory for the image of the window and pop it from the window stack. Typically things cannot be closed until they have been opened. See also *4GL window, close, cursor, file, open, screen form.*

column 1) In a database, a column is a data element containing a particular type of information common to every row of the table. In other database terminologies, a column is sometimes called a “field” or an “attribute.” See also *attribute, database, row, table.*

2) On a screen, a column is the x-coordinate of a particular position. The y-coordinate is called a row. Several 4GL statements use rows and columns in this sense to identify location of display. See also *row, screen.*

command line A line of text typed by the user at the operating system prompt to run a program. In a character-based environment, all programs are invoked by a command line with optional command-line arguments and options. See also *operating system.*

comment Descriptive information put in a source file to explain the file’s contents. Comments are introduced with special symbols to notify the compiler that subsequent text can be ignored. In 4GL source modules, comments can be introduced by putting the left brace ({) in the first position of the source file line to “open” a comment. You must then also include a right brace (}) anywhere on a line to “close” the comment. To put a comment on the same line as code, precede the comment with a double-hyphen (--)

or the “#” symbol. In 4GL form specification files, comments can be introduced with braces and the double-hyphen; the initial “#” symbol is not valid. See also *form specification file, header, source module.*

commit Successfully end a transaction by accepting all changes to the database since the beginning of the transaction. When the transaction is committed, all open database cursors (except hold cursors) are closed and all locks are released. The COMMIT WORK statement commits the current transaction. See also *cursor, log, roll back, transaction.*

compile 1) Translate a program from the source code written by the developer to a form executable by the computer (machine code). This translation is done by a system program called a “compiler.” Results of the translation are called “object code” or a “compilation.” See also *debug, execute, interpret, link, source module.*

2) In 4GL, you can compile 4GL source code into either p-code or C language code. For the p-code, the compiler translates the 4GL code into an intermediate form (p-code) that must be executed by the P-code runner. For C code, the compiler first calls a preprocessor to translate 4GL code into INFORMIX-ESQL/C code. It then translates the **ESQL/C** into object code, executable directly from a command line. See also *C Compiler version, command line, compiler directive, p-code, preprocessor, Rapid Development System.*

compiler directive An instruction within a source module to a compiler, as opposed to an executable *statement.* In the C language, directives can address the preprocessor portion of the compilation, requesting, for example, conditional compilation or inclusion of a named file. In 4GL, the WHENEVER and DEFER statements are compiler directive. The effect of the WHENEVER lasts until the end of the source file (or until overridden by another WHENEVER). See also *compile, exception handling, preprocessor, source module, statement.*

concatenate To form a character string by appending a second character string to the end of a first character string. In 4GL, the concatenation operator is a comma (,). See also *character, string, string operators.*

consistency checking The process of verifying that the user session’s collation and character set variable settings match settings in the database locale. In the Implicit and Explicit NLS environments, consistency checking determines whether or not access to a database is permitted. In the Open NLS environment, consistency checking is overridden. See also *Explicit NLS environment, Implicit NLS environment, NLS environment, Open NLS environment.*

constant A value which, unlike the value of a variable, does not change during execution of a program. Examples of 4GL constants are NOTFOUND, FALSE, and TRUE. The values of 4GL constants cannot be changed by the developer. See also *Boolean, literal, variable.*

control block A statement block that executes when a certain condition (the "activation clause"), becomes true. In 4GL, control blocks like BEFORE FIELD, AFTER INPUT, AFTER CONSTRUCT, and ON KEY occur in the user interaction statements; in this context they are often called "form management blocks." Control blocks also occur in the FORMAT section of a report, including PAGE HEADER, AFTER GROUP OF, and ON EVERY ROW. See also *activation key, report, statement block, user interaction statements.*

control character A character whose occurrence in a specific context initiates, modifies, or stops a control function (an operation to control a device, for example, in moving a cursor or in reading data). Control characters have values below ASCII 32 in the ASCII character set. In a 4GL program, some control characters have predefined functions (pressing CONTROL-W obtains on-line Help). The developer can also define actions that use CONTROL keys in conjunction with another key to execute some programming action. See also *activation key, ASCII, character, logical key, modifier key.*

current The one item, among many similar items, that is about to be or was most recently used. The current directory is the host system directory that was selected most recently (it is where files are first looked for). The current row is the row that was last fetched through a database cursor (it can be deleted or updated using the cursor). The current window is the window most recently activated (it receives your keystrokes). The current statement is the program statement being executed (it is displayed in an error message if the program terminates). See also *cursor, directory, row, statement.*

cursor 1) A focal point at which action can be applied. A text cursor is a mark showing the focal point for keyboard input. See also *keyboard focus, text cursor.*

2) A database cursor is an identifier associated with an active set. It points to the current row in the active set. This row can be fetched, deleted, or updated. 4GL supports the following types of database cursors: sequential, scroll, hold, update, and insert. See also *active set, close, current, identifier, open, prepared statement, query, row, scrolling.*

data conversion See *data type conversion.*

data entry 1) The action of providing, usually at the keyboard, data values to a computer program. Data entry is performed by the user of an application at run time. In a database application, these values are usually stored in a database. See also *application program, key, user interaction statements.*

2) A set of data values to be stored in program variables, and often, in a database table. The INPUT, INPUT ARRAY, and PROMPT statements accept data entry. See also *query criteria, table, user interaction statement.*

data file A file that contains the input used by a program. Data files are not executable; they contain data that is to be read or acted upon by other programs. See also *file.*

data type An interpretation to use on a stored value. In 4GL, database columns, program variables, form fields, and formal arguments of a function (or report) all have data types associated with them. The 4GL data types include: integer—SMALLINT, INTEGER; fixed-point—DECIMAL(*p,s*), MONEY; floating-point—DECIMAL(*p*), FLOAT, SMALLFLOAT; character—CHAR, VARCHAR; blob—BYTE, TEXT; time—DATE, DATETIME, INTERVAL; structured—ARRAY, RECORD. A column in a database can also use the SERIAL data type. See also *blob, character, data type conversion, declare, fixed-point number, floating-point number, integer, interval, operator, simple data type, structured data type, variable.*

data type conversion The process of interpreting and storing a value of one data type as some other data type; sometimes called simply "data conversion." The pairs of data types for which data conversion is possible without error are called "compatible data types." This process can be automatic or explicit. 4GL performs some automatic data conversion in expressions and assignment. It also provides facilities to perform some explicit data conversion (for example the EXTEND( ) function). In addition, the LIKE keyword, when used in a variable definition, provides "indirect data typing." See also *column, data type, define, indirect typing.*

database A collection of related data organized in a way to optimize access to this data. A "relational database" organizes data into tables, rows, and columns. Informix databases are relational databases. At run time, a separate database engine process is the portion of the database management system that actually manipulates the data in the database files. To access a database, a 4GL application must specify it with the DATABASE statement and must use SQL statements. See also *column, database engine, process, row, SQL, system catalog, table.*

database cursor See *cursor.*

database engine The part of a database management system that manipulates data files. This process receives SQL statements from the database application, parses them, optimizes the approach to the data, retrieves the data from the database, and returns the data to the application. Informix provides two database engines: INFORMIX-OnLine and INFORMIX-SE (both UNIX and DOS versions). The database engine is also called the "back end," "database server," or "database agent." See also *application development tool, database, process.*

database locale Locale of the user at the time of database creation, permanently saved in database system tables and consulted when the database is accessed. Currently only the collation (LC_COLLATE) and character set (LC_CTYPE) variables are saved. See also *Native Language Support, user locale.*

database server See *database engine.*

debug 1) Finding and removing run-time errors in a computer program. This analysis is often done by special software products called "debuggers." You can analyze a program to detect and locate errors in its logic, change the source code appropriately, then compile and run the program again. See also *compile, execute, link.*

2) If you are using the **RDS Version**, you can use the INFORMIX-4GL Interactive Debugger to debug 4GL programs. The Debugger helps you control and monitor program execution and inspect the application state. The Debugger is purchased separately from 4GL. See also *program execution, Rapid Development System.*

declare To make the name and data type of a variable known to a compiler. In 4GL, the DEFINE statement declares variables so the 4GL compiler can verify references to the variables in the succeeding code. The GLOBALS statement declares global variables. See also *compile, data type, define, global variable, variable.*

default The value that will be used, or the action that will be taken, unless you specify differently. In many SQL and 4GL statements, there is a default action that will be used if you do not specify another; for example, the FETCH statement retrieves the NEXT row unless you specify a different keyword such as PRIOR. Screen forms can specify a default value for input from each field, in case the user fails to enter one. See also *screen form, variable.*

define To allocate memory for storage of a variable. At run time, the DEFINE statement indicates how much storage should be allocated for a variable. The GLOBALS statement defines global variables. To define a function (or report) is to specify the actions performed by the function. See also *data type, declare, execute, function, global variable, report, variable.*

Delete key The logical key that the user can press within a 4GL application to delete the current line of a screen array (the current screen record) during the INPUT ARRAY statement. 4GL automatically deletes the associated line of the program array. By default, the physical key for Delete is F2. See also *Insert key, logical key, program array, screen array, screen record.*

delimiter A character that separates one unit of text from another. The eye can easily see boundaries based on context, but programs need unambiguous marker characters to detect the end of one item and the start of the next. In data produced by the UNLOAD statement, the data from each column ends with a delimiter (| by default) so that the LOAD command can recognize the end. In the form specification file, brackets ([ ]) mark or delimit the fields of the form. See also *form specification file.*

developer An individual who develops computer programs, taking them from design, coding, and debugging to general release. 4GL provides the developer with a means of developing database applications. Also referred to as the "programmer." See also *application program, user.*

development environment The special set of tools that a developer can use to develop computer programs. This environment may include text editors, language compilers, function libraries, program linkers, program debuggers, and other program utilities. Some of these tools may be accessed by "wrapper" programs, which are executive programs that decide which tools need to be run. The Programmer's Environment is an integrated development environment that combines many of these tools into a single, cooperative environment. See also *compile, debug, execute, link, Programmer's Environment, text editor.*

directory A directory is a "file folder;" it contains other files. Directories can also contain other directories; this is used to organize related files into categories. The user can construct a hierarchy of sub-directories and files that resembles an inverted tree in structure. Each user has a home directory that represents the top level of the user's personal hierarchy of other directories and files. The current directory is a single directory (selectable by the user) that enables the user to be "in" one directory at a time, and to refer to its files unambiguously. To refer to files in other directories, the user must supply a path, that is, a list of the sub-directories that describe the location of those files. See also *current, file, operating system.*

element See *array element.*

environment variable A special variable with a value that is maintained by the operating system and made available to all programs. They are usually stored in a special system area and contain system specifics such as the path (the directories in which the operating system looks for user-invoked commands). See also *operating system, shell.*

error An exception that indicates failure of a requested action or an illegal specification. Errors can occur during compilation—during preprocessing of program statements or during the linking stage—or during execution of the program. At run time, some errors are "fatal" in that the program cannot continue execution (run-time errors); others are recoverable in that the program can take corrective action and continue execution. Rounding errors can occur during truncation in rounding; overflow errors can occur during arithmetic operations in which the size of the result is larger than the size of the space in which the result is to be stored. See also *compile, error handling, exception, execute, link, status variable, truncation.*

error handling Program code that checks for a run-time error. By default, 4GL terminates a program when it encounters a run-time error. The 4GL WHENEVER ERROR statement can change this default error handling behavior. With the WHENEVER ERROR CONTINUE statement, 4GL sets the built-in **status** variable to a negative value and continues execution when it encounters a run-time error. The program must explicitly check the value of **status** and determine appropriate action. See also *error, error log, error text, exception handling, program execution, status variable.*

error log A file that receives a program's error information at run time. 4GL contains some built-in functions that allow you to make entries in the error log: ERR_GET( ), ERR_PRINT( ), ERR_QUIT( ), ERRORLOG( ), and STARTLOG( ). See also *built-in function, error, error handling, status variable.*

error message Text that describes a 4GL error. Each error is identified by an integer, usually negative, called an "error code." Each code corresponds to a specific error message. Such messages can be retrieved by running the **finderr** utility or within a program by making a call to the ERR_GET( ), ERR_PRINT( ), or ERR_QUIT( ) built-in functions. By default, 4GL automatically displays some run-time error messages on the screen. See also *error handling, error log, status variable.*

escape character A character which indicates that the following character, normally interpreted by a program, is to be printed as a literal character instead. Usually programs which handle user input (such as text editors and shells) have some characters which have special interpreted meanings. The escape

character is used in conjunction with the interpreted character to "escape" or ignore the interpreted meaning. The ASCII escape character is a non-printing character with value of ASCII 27. In 4GL, the backlash (\ ) symbol can be used as an escape character: the string "\\" would indicate that the backslash character is to be sent. See also *ASCII, character, printable character, shell, text editor.*

Escape key The physical key usually marked ESC on the keyboard. It sends the ASCII code for the escape character. This key is the default Accept key in user interaction statements like CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY. See also *Accept key, escape character, key, user interaction statement.*

exception An event which occurs at run time for which the program may want to check. Exceptions in 4GL include: run-time errors, (an error returned by a database server, a state initiated by a stored procedure statement, or an error detected by the database application program), an unsuccessful database query (**status** variable is set to NOTFOUND), and warnings (SQL conditions), and the pressing of the Interrupt or Quit key. See also *error, exception handling, Interrupt key, Quit key, status variable, warning.*

exception handling Program code that checks for exceptions and performs recovery actions in the event they occur. By default, 4GL performs the following exception handling: run-time errors—terminate the program; SQL NOTFOUND—set **status** to NOTFOUND and continue execution; warnings—continue execution; Interrupt (or Quit) key—terminate program. The developer can change the default exception handling for these first three types of exceptions with the WHENEVER statement: run-time errors —WHENEVER ERROR; SQL NOTFOUND—WHENEVER NOT FOUND; warnings—WHENEVER WARNING. The 4GL DEFER statement changes handling of the Interrupt and Quit keys. See also *error handling, Interrupt key, program execution, Quit key, status, warning.*

executable file 1) A file containing machine code (in binary form) that has been linked and is ready to be run on a computer. May also refer to a collection of instructions that can be executed by a command interpreter or processor, for example a batch file. See also *compile, execute, file, interpret, link.*

2) In 4GL, an executable file can be either interpretive p-code (with a **.4gi** or **.4go** extension) or compiled C code (with a **.exe** extension), depending on the version of 4GL you are using. See also *command line, p-code.*

executable statement A statement that requires processing action at run time. Executable statements are distinguished from declarative statements (that provide information about the nature of the data without themselves causing any

processing) and compiler directives which are instructions to the compiler. All 4GL statements are executable except: MAIN, DEFINE, DEFER, FUNCTION, GLOBALS, REPORT, and WHENEVER. See also *compiler directive, declare, define, statement.*

execute 1) Run a compiled or interpreted program by carrying out the instructions in an executable file. To execute or run a program, the operating system must create a process for the program and then allocate the CPU (and any other resources needed by the program) to this process. The state of the program in execution is often called "run time." See also *compile, debug, executable file, interpret, link, operating system, process, resources.*

2) A 4GL executable file contains either interpreted p-code or compiled C code, depending on the version of 4GL you are using. See also *C Compiler Version, command line, Rapid Development System.*

execution stack See *call stack.*

Explicit NLS environment An Informix tool running in an environment where DBNLS is set to 1 and COLLCHAR unset. **INFORMIX-4GL** programs cannot use the Explicit environment. The Explicit environment supports both CHAR and NCHAR data types. CHAR (and VARCHAR) data sort and compare according to US English ASCII, whereas NCHAR (and NVARCHAR) data sort and collate according to collation and character set settings of the database locale. See also *Implicit NLS environment, NLS environment, Open NLS environment.*

explicit transaction A transaction that the developer must explicitly begin and end. The BEGIN WORK statement indicates the beginning of the transaction. The developer must explicitly indicate the end of the transaction with the COMMIT WORK and ROLLBACK WORK statement. A database that is not ANSI-compliant but that has a transaction log uses explicit transactions. See also *ANSI-compliant, commit, roll back, transaction.*

exponent 1) In the representation of a FLOAT or SMALLFLOAT value, a signed integer indicating the power to which the mantissa is to be raised. See also *floating-point number, mantissa.*

2) The right-hand unsigned integer operand of the exponentiation (**) operator in 4GL expressions. See also *arithmetic operators.*

expression A sequence of operators, operands, and parentheses that can be evaluated to a single value, usually at run time. In 4GL, an expression should evaluate to a simple 4GL data type: number (Boolean, integer, floating-point, and fixed-point), character, or time (DATE, DATETIME, and INTERVAL). See also *Boolean, character, data type, fixed-point, floating-point, integer, interval, operand, operator, precedence, regular expression.*

external signal The notification of an operating system event that is delivered to a process. 4GL programs can handle two external signals: interrupt and quit. See also *Interrupt key, operating system, process, Quit key.*

field 1) An area for holding a data value. Usually refers to a delimited, unprotected area on a screen form used for entry and display of a data value. Such fields can have "field attributes" associated with them that control, for example, the case of letters, default values, and so on. When a form is active, the location of the cursor designates what is called the "current field," the field in which the user can enter data. A field on a screen form corresponds to a column in a database, unless it has been defined as a "form-only field." See also *attribute, current, form, form specification file, form-only field, multiple-segment field, screen array, screen form.*

2) In some database terminologies, the term used for a "column." See also *column.*

field buffer A portion of computer memory associated with and holding the current contents of a particular screen field. In 4GL, the GET_FLDBUF( ) built-in function allows the developer to examine the field buffer. See also *built-in function, field.*

field tag A unique name identifying a field in the SCREEN section of a form specification file. It is also used in the ATTRIBUTES section to assign a set of field attributes to the field. Unlike a label, a field tag does not appear when the form is displayed. See also *attribute, field, form specification file, label.*

file 1) A named collection of information stored together, usually on disk. A file can contain the words in a letter or report, a computer program, or a listing of data. Files are usually stored in directories and are managed by the operating system. See also *data file, directory, executable file, form specification file, help file, log, operating system, output file, source module.*

2) In some database terminologies, the term used for a "table." See also *table.*

filename extension The part of the filename following the period (.). It usually identifies the purpose of the file. For example, form specification files have a **.per** extension while compiled form files have a **.frm** extension. 4GL source modules have a **.4gl** extension. INFORMIX-ESQL/C files have a **.ec** extension. See also *executable file, file, form specification file, help file, source file.*

fill character Specific ASCII characters that are used to provide formatting instructions in a format string. In 4GL reports, the ampersand (&) instructs the PRINT statement to insert leading zeros when a number does not completely fill the format string. Fill characters are used in report and forms. See also *ASCII, character, form specification file, format string, report.*

fixed-point number A real number with a fixed scale. In 4GL, the fixed-point number data types are DECIMAL(*p,s*) and MONEY. These data types store values that include a fractional part as fixed-point numbers. See also *data type, floating-point number, scale, simple data type.*

flag 1) A value used to indicate or "flag" some condition. You can define a program variable as a flag (usually assigning it the values TRUE and FALSE). See also *Boolean, variable.*

2) A command-line option to an operating system program is sometimes called a flag as well. See also *command line, operating system.*

floating-point number A real number with a fixed precision and undefined scale. The decimal point "floats" as needed to represent an assigned value. In 4GL, the floating-point number data types are FLOAT, SMALLFLOAT, and DECIMAL(*p*). See also *data type, exponent, fixed-point number, mantissa, precision, scale, simple data type.*

foreground process A process that currently has access to a window or the screen. It requires this access because it needs to perform input or output. See also *background process, process, screen.*

form See *screen form.*

form field A field on a screen form. See *field.*

form specification file An ASCII source file that describes the logical format of the screen form and provides instructions about how to display data values in the form at run time. You define a screen form in its source file (with a **.per** extension) and create a compiled version (with a **.frm** extension) for use at run time. **form4gl** creates the compiled version. Sometimes this file is referred to as simply "form specification." See also *field, field tag, file, screen array, screen form.*

form-only field A field on a screen form that is not associated with any database column. Usually, a form-only field is used for display purposes only. See also *column, field, form specification file, screen form.*

formal argument In a function definition, the variable in the argument list that serves as a placeholder for an actual argument. The argument list determines the number and data types of the function's arguments. In the function call, the actual argument sends the value to the function. See also *actual argument, calling routine, function call, formal argument.*

format string A quoted string whose characters specify how to display data values. The USING operator, the FORMAT and PICTURE field attributes, and certain environment variables can use format strings. See also *attribute, environment variable, fill character, quoted string, string.*

Formatted mode An output mode of a 4GL program in which screen addressing is used. 4GL enters this mode when it executes any 4GL user interaction or output statement (ERROR, MESSAGE, DISPLAY AT, and so on) except a simple DISPLAY statement (one without an AT, BY NAME, or TO clause). Output in Formatted mode displays in the 4GL screen. It should not be mixed with Line mode. See also *Line mode.*

fourth-generation language 1) A programming language, closely approximating a "natural language," designed and developed for a particular class of applications. Because they focus on a specific type of application, such languages can anticipate the types of actions these programs need to perform. As a result, many typical operations can be encapsulated into a single but powerful statement. Sometimes abbreviated as 4GL.

2) 4GL is a fourth-generation language for the creation of database applications. It includes the ability to embed SQL statements in a program as well as providing additional statements, operators, and functions to assist in the creation of database applications. See also *application program, built-in function, built-in operator, database, operator, SQL, statement.*

function 1) A named collection of statements defined to perform an application task, often one which needs to be repeated. Functions can be defined to accept arguments and to return values. Within 4GL, the following functions exist: programmer-defined function—function written in 4GL by the developer; C function—written in C by the developer; **ESQL/C** function—function built into the INFORMIX-ESQL/C product for use in a C program written by the developer; 4GL built-in function—written by Informix as part of the 4GL language to enable the developer to perform specific tasks within 4GL; built-in SQL function—written as part of the SQL access language. See also *argument, built-in function, programmer-defined function, return value, SQL, statement.*

2) A 4GL function (program block defined with the FUNCTION statement) is often referred to simply as a "function." See also *4GL function.*

function call The invocation, by a calling routine, of a programmer-defined function. This syntax includes the function name followed by the actual argument values, in parentheses. The function call can be explicit (with the CALL statement) or implicit (by including the function call within a 4GL expression). See also *actual argument, calling routine, programmer-defined function, pass-by-reference, pass-by-value.*

function definition See *4GL function.*

function key A key named "F#" where "#" is a number. Most keyboards have functions keys F1 through F12. In 4GL, the developer can define actions to perform when the user presses a certain function key. These actions are defined with the ON KEY clause of the CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or PROMPT statement, or in the KEY clause of the MENU statement. Here and in the OPTIONS statement, you can use the notation F1 through F64 to denote the individual function keys (but not all keyboards support that many function keys). See also *control block, key, logical key.*

global variable A variable defined outside all program blocks and accessible within all program blocks of the program. The scope of a global variable is all statements that follow the global variable definition and all source modules which reference the global variable's definition. In 4GL, global variables are defined within the GLOBALS statement either at the top of a source module (outside all program blocks), or in a separate source file. See also *declare, define, program block, scope of reference, source module, variable.*

header A comment block at the top of an ASCII file that identifies the file contents. Contents can include the purpose, author, modification information, and other relevant information. Headers are often used in 4GL source modules and form specification files. See also *comment, form specification file, source module.*

help file A file containing help messages for a 4GL application. Text associated with each message must be uniquely identified within a given help file. You define a help message in its source file (by default having a **.msg** extension) and create a compiled version (default **.iem** extension) for use at run time. The **mkmessage** utility creates the compiled version. Sometimes this file is referred to as a "help message" file. See also *file, help message.*

Help key The logical key that the user can press in a 4GL application to display a help message for the current form, field, or menu option. By default, the physical key for Help is CONTROL-W. See also *logical key, help file, help message.*

help message Text that provides information and guidelines on a particular topic or for a particular context. In 4GL, each message is identified by a positive, non-zero integer, called a "help number." Each number corresponds to one and only one help message text resident in the currently designated help file. Help messages can be displayed automatically or at the request of the user (the Help key). See also *Help key, help file.*

hexadecimal number A number represented in base 16. The right-most digit is multiplied by 16 to the power of zero. The digit immediately to the left is multiplied by 16 to the first power. The digit immediately to the left of that is multiplied by 16 to the second power, and so on. The characters that represent a hexadecimal number are 0-9 and A-F (for 10-15).

identifier A sequence of letters, digits, and symbols which the compiler recognizes as a programmer-defined name of some entity. In 4GL, an identifier can include letters, digits, and underscore symbols (_). It must have either a letter or an underscore (_) as the first character. It can be up to 50 characters in length but the first seven must be unique among similar program entities that have the same scope of reference. 4GL does not distinguish between upper and lowercase letters. Types of identifiers include variable names, cursor names, function names, report names, table names, window names, form names, prepared statement names, and report names. See also *4GL function, 4GL window, case sensitivity, cursor, keyword, name space, prepared statement, report, reserved word, scope of reference, screen form, table, variable.*

Implicit NLS environment Defined as DBNLS set to 1 and COLLCHAR set to 1. A tool operating in the Implicit NLS environment produces NLS applications in which all character data is sorted and compared according to the locale established at the time of database creation. The tool user cannot create character data columns that sort in US English if the locale the database was created in was non-US-English. All character columns defined by tool in the Implicit environment are defined within the tool as type CHAR (or VARCHAR, if variable-length), but interpreted by the server as type NCHAR (or NVARCHAR) if the database is non-US-English. This is the recommended environment for access and creation of NLS databases. See also *Explicit NLS environment, NLS environment, Open NLS environment.*

implicit transaction A transaction that automatically begins when an SQL statement which alters the database executes. The developer must explicitly indicate the end of the transaction with the COMMIT WORK and ROLLBACK WORK statement. An ANSI-compliant database uses implicit transactions. See also *ANSI-compliant, commit, roll back, transaction.*

index 1) A database file containing a list of unique data values, with pointers to the database rows that contain those values. Indexes are used to reduce the time required to order rows and to optimize the performance of database queries. See also *database, query, row.*

2) A subscript value into an array. See also *array, subscript.*

indirect typing The process of assigning a data type to a variable by referencing a database table or column. In 4GL, indirect typing is carried out by the keyword LIKE in a variable definition. See also *column, data type conversion, define, table, variable.*

Informix-defined A classification describing environment variables which did not originate in the X-Open Portability Guide version 3 (XPG3) standard but are relevant to NLS. This includes DBNLS, COLLCHAR, DBAPICODE, DBLANG, DBFORM, DBFORMAT, DBMONEY, and DBDATE. Informix-defined variables are consistent in syntax and meaning across platforms. This is in contrast to X-Open defined variables, which rely on facilities provided by the computer manufacturer, and can vary from system to system. See also *environment variables, XPG3.*

input record A group of related values that are passed to a 4GL report for formatting. The report formats data one input record at a time. The OUTPUT TO REPORT statement sends an input record to a report. See also *report.*

Insert key The logical key that the user can press in a 4GL application to insert a new line at the current position of the screen array during the INPUT ARRAY statement. 4GL automatically inserts a line at the associated position of the program array. By default, the physical key for Insert is F1. See also *Delete key, logical key, program array, screen array.*

int_flag variable See *Interrupt key.*

integer 1) A real number that has no fractional part. In 4GL, the INTEGER and SMALLINT data types can store integer values within the limits of their ranges. Boolean values are also stored as integers. See also *Boolean, data type, simple data type.*

2) In uppercase, INTEGER is a data type for storing integers whose absolute value is no greater than 2,147,483,647.

interactive A mode of execution in which a program accepts input from a user and processes that input, a program that sends output to the screen, or a program that does both. 4GL programs which use user interactive statements are interactive. See also *batch, program execution, user interaction statement.*

interpret 1) Run a program that has been compiled to intermediate code. The executable file contains instructions in intermediate code. This translation is done by a system program called an "interpreter," sometimes called a "runner." See also *compile, debug, execute, link.*

2) The **RDS Version** of 4GL can interpret 4GL source code by executing the intermediate code (the p-code) produced by the 4GL compiler. See also *p-code, Rapid Development System.*

Interrupt key The logical key that the user can press within a 4GL application to indicate cancellation of the user interaction statement. If the 4GL program does not include the DEFER INTERRUPT statement, pressing this key terminates program execution. With DEFER INTERRUPT, pressing the Interrupt key sets the built-in **int_flag** variable to TRUE and cancels the current interaction statement, resuming execution at the next 4GL statement. However, further response to the Interrupt signal may be deferred until the next pause for user input. The physical key for Interrupt is CONTROL-C. See also *Accept key, exception handling, interrupt signal, logical key, Quit key, user interaction statement.*

interrupt signal A high-priority kind of signal sent to a running program either by the operating system directly or by the user. An interrupt signal is usually a command to interrupt a running program. In a 4GL application, the user may invoke an interrupt signal by pressing the Interrupt key. See also *Interrupt key, program execution.*

interval 1) A span of time. In 4GL, there are two types of intervals, namely *year-month* (those measured in years and months), and *day-time* (those measured in smaller time units: days, hours, minutes, seconds, and fraction of seconds). See also *data type, simple data type.*

2) In uppercase, INTERVAL is a data type for intervals of time.

key 1) In database terminology, a *key* value is that part of a row that makes the row unique from all other rows; for example, a SERIAL number. At least one such value must exist in any row; the most important is designated as the "primary key." See also *column, rowid, table.*

2) In application terminology, when speaking of the keyboard, a *key* is what the user presses to enter text or commands in the application. The actual keys of the keyboard (a, ESCAPE, RETURN) are often called *physical* keys to distinguish them from *logical* keys (which may be made up of a sequence of keys). See also *accelerator key, activation key, control character, function key, keyboard, logical key, mnemonic key, modifier key, RETURN key.*

keyboard focus The area on the screen that currently gets input from the keyboard; for example, a particular text field. The keyboard focus is usually indicated by highlighting or the presence of a text cursor. See also *active window, key, screen, text cursor.*

keyword A sequence of letters recognized by a compiler as having a reserved meaning within a language. In 4GL, examples of keywords are INPUT, INSERT, TO, and LIKE. In 4GL documentation, keywords are shown in uppercase to improve readability. However, 4GL keywords are not case-specific. See also *case sensitivity, identifier, reserved word.*

label A character string used as a point of reference. For example, a label on a screen form helps to identify a form field when the form displays. In a 4GL LABEL statement, the label is the identifier which indicates the position in a 4GL program to which GOTO statements can transfer control. See also *screen form, string.*

language and formatting variable An environment variable that controls aspects of the language and formatting environment. These include LANG, the LC_ variables, DBFORMAT, DBMONEY and DBDATE. This classification is in contrast with meta-environment variables, which control aspects of the NLS meta-environment, such as whether or not NLS is active or whether or not implicit mapping is turned on. See also *environment variable.*

language supplement A product obtained from an Informix sales office that provides settings for an additional national language when installed in a 6.0 server environment. See also *locale file, Native Language Support.*

LC_ variable Any of the four environment variables LC_COLLATE, LC_CTYPE, LC_MONETARY, or LC_NUMERIC that begin with the letters LC followed by the underbar (_) symbol. See also *environment variable, Native Language Support.*

Line mode An output mode of a 4GL program in which screen addressing is not used. 4GL enters this mode and displays the Line mode overlay when it executes a simple DISPLAY statement (one without an AT, BY NAME, or TO clause).4GL remains in Line mode as long as it encounters additional simple DISPLAY statements or the PROMPT statement. When it encounters any other output statement (ERROR, DISPLAY AT, and so on) or a user interaction statement, 4GL returns to Formatted mode. Because this mode results in a simple stream of characters to standard output, it should not be mixed with Formatted mode. See also *Formatted mode, Line mode overlay.*

Line mode overlay A window that overlays the entire 4GL screen when 4GL enters Line mode. The Line mode overlay remains as long as the program remains in Line mode. It is only updated, however, when 4GL executes either a PROMPT or a SLEEP statement. See also *4GL window, Formatted mode, Line mode.*

link 1) Combine one or more source modules (which have been compiled separately) into a single executable file or program. This merging is done by a system program called a "linker" or a "link editor." The linker verifies it can locate all functions called and all variables used. See also *compile, debug, execute, executable file, source module.*

2) The 4GL Programmer's Environment can link compiled 4GL source modules into a single executable file. The Programmer's Environment can handle the entire process of compilation, linking, and running of a multi-module 4GL program. If you do not use the Programmer's Environment, you must explicitly link compiled 4GL source modules into a single executable file. See also *p-code, Programmer's Environment.*

literal A character constant. In the format string of a 4GL PICTURE field attribute, for example, any characters except "A", "#", and "X" are literals because they are displayed unchanged in the format string. In 4GL statements, literal strings must be surrounded by quotation marks (" "). See also *attribute, constant, form specification file, quoted string, string.*

local variable A variable defined within a program block. The scope of a local variable is limited to those statements within the program block. In 4GL, local variables are defined within MAIN, FUNCTION, or REPORT program blocks with the DEFINE statement. See also *declare, define, program block, scope of reference, variable.*

locale file A file installed on the system which specifies language or formatting behavior for one or more settings of one or more LC_ variables. The format, naming, and use by the system of locale files varies between computer manufacturers. Locale files are installed by installing language supplements. See also *language supplement.*

locale-sorted Character data which sorts in the order specified by the LC_COLLATE environment variable. Corresponds at the server to the data types NCHAR and NVARCHAR. See also *Native Language Support.*

log 1) With an INFORMIX-OnLine database engine, a *physical log* contains images of entire pages before they were changed. Physical logs are used during fast recovery when INFORMIX-OnLine is coming up. See also *error log, file.*

2) A *logical log*, sometimes called a "transaction log," records changes performed on a database during the period the log was active. A logical log includes, as needed, images of the row before it was changed and images of the row after it was changed. Logical logs are used to roll back transactions, recover from system failures, and restore databases from archives. See also *commit, roll back, transaction.*

3) The 4GL STARTLOG() function can specify an *error log* file in which to record run-time errors.

logical key A key that the user can press to perform certain tasks predefined by 4GL or the operating system. These keys include the following: Accept key, Delete key, Help key, Insert key, Interrupt key, Quit key. Each key is associated with some default physical key. The OPTIONS statement can assign most logical keys to different physical keys. Certain 4GL statements can reference logical keys with keywords like ACCEPT, DELETE, INSERT, and so forth. See also *Accept key, Delete key, Help key, Insert key, Interrupt key, key, Quit key.*

login The procedure that identifies a user to a computer. If the login is successful, the user is granted access to the system. See also *user name.*

main menu The top menu in a hierarchy of nested menus. See also *menu, ring menu.*

MAIN program block The program block that 4GL begins executing when it starts a 4GL program. This program block is defined with the MAIN statement and includes all statements between the MAIN and the END MAIN keywords. When it reaches the END MAIN keywords, 4GL ends the program. See also *4GL function, function, normal execution, program block, report.*

mantissa 1) In the representation of a FLOAT or SMALLFLOAT value, a signed integer indicating the number that is to be raised to the power indicated by the exponent. See also *exponent, floating-point number.*

2) The left-hand unsigned integer operand of the exponentiation (**) operator in 4GL expressions. See also *arithmetic operators.*

member See *record member.*

menu A visual object from which the user can choose one of several options, called "menu items." Menus often control a program by providing menu items for actions that can be performed. See also *main menu, ring menu.*

menu option A choice the user can make from a ring menu. A menu option can be visible, in which case it appears in the ring menu. It can also be invisible, in which case the user must know the correct activation key for choosing the

option. A hidden menu option cannot be activated by the user unless the program uses SHOW MENU within the MENU statement. See also *activation key, ring menu.*

meta-environment variable An environment variable that controls aspects of the NLS meta-environment, such as whether or not NLS is active or whether or not implicit mapping is turned on. These include DBNLS, COLLCHAR, DBAPICODE, DBLANG and DBFORM. This classification is in contrast with NLS environment variables that control the language and formatting environment.

mnemonic key A shorthand name for a key, menu option, or command. See also *activation key, alias, key.*

MODE ANSI See *ANSI-compliant.*

modifier key A key that is held while pressing another key in order to modify its meaning. See also *control character, key.*

module A group of related functions. If these related functions share variables, these variables can be defined as module variables. During program execution, the current module is the source file that contains the program block currently being executed. See also *current, module variable, source module.*

module variable A variable defined outside all program blocks. The scope of reference of a module variable is all statements that follow its definition. In 4GL, module variables are defined with the DEFINE statement at the top of a source module, outside all program blocks. See also *declare, define, program block, scope of reference, source module, variable.*

multiple-segment field In a screen form, a field consisting of several, separately delimited parts, each sharing a common field tag. Such a field allows long character strings to be displayed or entered on successive lines of the form. A multiple-segment field requires the WORDWRAP field attribute in the form specification file. See also *field, field tag, form specification file, screen form.*

name space The set of identifiers of different types whose names must be unique within the same scope. For example, the following types of identifiers have the same name space: cursor names, window names, form names, function names, global variable names, and report names. Because they share the same name space, none of them can have the same name: a cursor cannot have the same name as a window or a global variable; however, a cursor could have the same name as a local variable as long as the cursor did not appear within the same scope as the local variable. See also *identifier, naming conventions, scope of reference, variable.*

naming conventions A set of rules for the creation of identifier names that assist in the recognition of the purpose of the identifier from its name. For example, prefixes could be used to identify: names of cursors (c_), windows (w_), program records (p_), program arrays (pa_), global variables (g_), screen arrays (sa_), and so on. See also *identifier, name space, scope of reference, variable.*

Native Language Support Based on the X/Open Portability Guide Version 3 (XPG3) standard, it specifies a means for localization of software to European geographical regions without the need for alteration to user applications. NLS is available only on UNIX systems that support X/Open NLS libraries. See also *ASCII, NLS database, NLS environment, XPG3.*

navigation Traversing fields or other controls within a window. You navigate by using the TAB and arrow keys. See also *key.*

NLS Acronym for Native Language Support.

NLS database An NLS database is a database created with NLS environment settings active (DBNLS set to 1 or 2). See also *NLS environment.*

NLS environment A combination of user environment variable settings in which DBNLS is set to 1 or 2. There are three NLS environments: Implicit, Explicit and Open. Different NLS environments are selected by way of different combinations of the DBNLS and COLLCHAR environment variables. In NLS environments: 1) the LANG and LC_ variables are considered in operations, 2) collation is specified by LC_COLLATE unless the column is type NCHAR and the environment is Explicit, 3) user defined names can contain any characters contained in the character set specified by LC_CTYPE, 4) non-NLS databases can be accessed but not created, 5) LC_COLLATE and LC_CTYPE values are saved in system tables at database creation time. See also *Native Language Support.*

Non-NLS database A non-NLS database is a database created in a pre-6.0 server environment, or in the Non-NLS environment. It can be accessed while an NLS environment is active, but LANG and the LC_ variables have no effect. See also *NLS database.*

Non-NLS environment A Non-NLS environment is defined as DBNLS unset or set to zero. In this environment databases created are non-NLS databases, NLS databases cannot be accessed, LANG and the LC_ variables have no effect, character collation order is US English, only ASCII characters may be used in identifiers, and default monetary and numeric formats are ANSI compliant. See also *NLS environment.*

normal termination The termination of the 4GL application by exiting the MAIN program block at the END MAIN keywords or with the RETURN statement. In the INFORMIX-4GL Interactive Debugger, you can no longer inspect the application state after normal termination because there are no active functions. See also *abnormal termination, active function, debug, MAIN program block, program execution.*

null value 1) A value that means "not known" or "not applicable." A null value is distinct from a string of blanks or from a value of zero. Database columns and program variables can have null values. In 4GL, this marker referred to by the keyword NULL. To test for a null value, use the IS NULL and IS NOT NULL operators. See also *Boolean operators.*

2) In other contexts, "null" is casually used to mean "empty"; for example, a character string with zero length is sometimes called "the null string." This can lead to confusion because an empty string (the string " ") has a specific non-null value, distinct from a NULL string. An empty string has a definite length (zero) while a NULL character value has an unknown length and value. See also *blank space, string.*

open To prepare something for use. In programming, opening something often entails allocating memory and other resources to it, and sometimes means getting exclusive access. Typically, things cannot be used until they have been opened; then they remain usable until they are closed. To open a cursor means to have the database engine process the query up to the point of locating the first selected row; this can entail significant processing and space in memory and on disk. To open a file is to locate the file and bring it into memory. To open a form is to find the compiled form file, bring it into memory, and prepare to display it. To open a 4GL window is to allocate memory for the image of the window, push it onto the window stack, and to display it on the screen. See also *4GL window, close, cursor, file, query, resources, screen form.*

Open NLS environment Defined as DBNLS set to 2 and COLLCHAR set to 1. Third party tools can use the Open NLS environment to access NLS databases by way of SQL commands to the database engine. The tool sends queries to the server for processing and gets back results that are properly sorted from the standpoint of the database locale, without the tool knowing what locale it is accessing. The Open NLS environment can also be used to perform LOAD and UNLOAD operations between locales. See also *Explicit NLS environment, Implicit NLS environment, NLS environment.*

operand A value on which an operation is performed. An operand can be a variable, a constant, or an expression. See also *constant, expression, operator, variable.*

operating system The software that provides an interface between application programs and hardware. It is the part of a computer system that makes it possible for the user to interact with the computer. It manages processes by allocating the resources they need. See also *command line*, *execute*, *process*, *resources*.

operator A symbol or keyword built into a language that returns a value from the values of its operand(s). Operators can generate a value from a single value (unary operators) or from two values (binary operators). See also *arithmetic operators*, *assign*, *associativity*, *binary operator*, *Boolean operators*, *built-in operator*, *operand*, *precedence*, *relational operators*, *string operators*, *unary operators*.

output file A file in which the results of a query or a report are stored. See also *file*, *query*, *report*.

owner A designation that associates an individual with a file or set of files. Informix databases can use ownership to restrict access to certain columns or tables. On UNIX systems, ownership also applies to files and directories for the purpose of limiting access to their contents and location within the file system. See also *database*, *operating system*.

p-code Abbreviation for pseudo-code. P-code is an intermediate form of code generated by the **RDS Version** of 4GL. Although p-code takes more memory to run, it is machine-independent. See also *compile*, *debug*, *execute*, *interpret*, *link*.

page A unit of data analogous to the page of a book. One page of a program array is the number of rows that can be displayed in the screen array at one time. The database engine stores data in pages. See also *program array*, *screen array*.

page header The top part of a page in a report. A "running header" appears at the top of each page of a report. Information (for example, the title and date) printed at the top of each page of a report is formatted in the PAGE HEADER and FIRST PAGE HEADER control blocks of a report. See also *control block*, *page trailer*, *report*.

page trailer The bottom part of a page in a report. Information (for example, the page number) printed at the bottom of each page of a report is formatted in the PAGE TRAILER control block. A page trailer is also referred to as a "footer." See also *control block*, *page header*, *report*.

pass-by-reference A method used in a function call that determines how an argument is passed to the programmer-defined function. With pass-by-reference, the address in memory of the actual argument is passed to the function. This method means that changes made to the value of the formal argument

within the body of the function will be visible from the calling routine when the function exits. 4GL uses pass-by-reference only for blob (BYTE and TEXT) variables. See also ***argument, blob, function call, pass-by-value.***

pass-by-value A method used in a function call that determines how an argument is passed to the programmer-defined function. With pass-by-value, the actual argument is evaluated and the resulting value is passed to the function. This method means that changes made to the value of the formal argument within the body of the function will not be visible from the calling routine when the function exits. 4GL uses pass-by-value for variables of all data types except blob (BYTE and TEXT). See also ***argument, blob, data type, function call, pass-by-reference.***

pathname The list of directories needed to identify a file within a directory hierarchy. In UNIX, directories of a pathname are separated by the slash (/). A file can be referred in two ways: by its absolute pathname—all directories starting from the root (top) of the directory hierarchy; by its relative pathname—the directories relative to the current directory. See also ***current, directory, file.***

pipe A connection of one process to another process such that the output of the first process is sent directly as input to the second process. It is one of several ways in which processes can communicate. It is common to speak of a process piping some data to another process. See also ***process.***

popup window A 4GL window that automatically appears when a predefined condition or event occurs. In a 4GL application, a popup window often contains a list of values for a particular field. The user can choose from this list rather than needing to type in the value directly. See also ***4GL window.***

precedence of operators The hierarchy of operators. It determines the order in which 4GL evaluates operators within an expression. 4GL evaluates higher precedence operators before those of lower precedence. The order in which operators at the same precedence level are evaluated is left to right. Precedence order can be changed by surrounding expressions with parentheses. See also ***associativity, expression, operator.***

precision The total number of significant digits in the representation of a numeric value or in a data type specification. The number 3.14 has a precision of three. See also ***floating-point number, scale.***

prepared statement The executable form of an SQL statement. SQL statements can be executed dynamically by creating character strings with the text of the statement. This character string must then be "prepared" with the PREPARE state-

ment. The result of the PREPARE is a prepared statement. The prepared statement can then be executed with the EXECUTE or DECLARE statements. See also *query by example*, *SQL*, *statement*, *statement identifier*.

preprocessor A program that translates "macro" code into statements conforming to the host language. The results of preprocessing can then be passed to a standard language compiler, such as C or COBOL. When using the **C Compiler Version** of 4GL, the compiler first sends the 4GL source module through a preprocessor to translate SQL statements into INFORMIX-ESQL/C calls before passing the file to a C compiler. See also *compile*, *source module*.

print position The logical location of the print head. A print position can be compared to a screen cursor in that both refer to a specific x,y coordinate on the page or screen. See also *column*, *report*, *row*.

printable character A character that can be displayed on a terminal or printer. These characters include the ASCII codes 32 through 126: A-Z, a-z, 0-9, symbols (!, #, $, ;, *, and so on), TAB (CONTROL-I), NEWLINE (CONTROL-J), FORMFEED (CONTROL-L) and the blank space character (   ). See also *ASCII*, *blank space*, *character*.

process An independent unit of operating system execution. It keeps track of the state of execution for a program. The operating system creates a process for each program being executed. It allocates resources needed by a program (memory, disk, CPU) to its process. The current process is the one that has been allocated use of the CPU. A 4GL application usually runs with two processes: the 4GL application (the "front end") and a database engine (the "back end"). See also *application development tool*, *background process*, *database engine*, *foreground process*, *operating system*, *pipe*, *resources*, *shell*.

program array A 4GL variable defined with the ARRAY keyword. A common use for a program array is as an array of records to store information to be displayed in a screen array. The DISPLAY ARRAY and INPUT ARRAY statements can manipulate program array values or records within the screen array. See also *array*, *program record*, *screen array*, *structured data type*, *variable*.

program block A programmer-defined group of 4GL statements that has its own scope during execution. The scope includes definitions and values of variables and may include arguments and return values. 4GL has the following program blocks: a MAIN program block, 4GL functions, and reports. Every executable statement must appear within some program block. Program blocks can neither overlap nor be nested. Any variable defined within a

program block is local to that block. See also *4GL function, call stack, executable statement, MAIN program block, programmer-defined function, report, scope of reference, statement block, variable.*

program execution The process of running (executing) a program. A program can be in the following states: running, suspended (by the system or the program itself) or terminated (abnormally or normally). See also *abnormal termination, debug, execute, normal termination.*

program record A 4GL variable defined with the RECORD keyword. A common use for a program record is for storing information in a screen record, a row of a table, or in a line of a screen array. See also *program array, record, row, screen record, structured data type, variable.*

program design database A database that describes the resources needed to create various executable programs. It is called **syspg4gl** (by default), and is accessed by the Programmer's Environment. Regardless of the version of 4GL you are using, (**RDS Version** or **C Compiler Version**), this database tracks for each 4GL program such resources as source files and compiler options. See also *executable file, Programmer's Environment.*

programmer-defined function A function written in 4GL that can be called in a 4GL program. The developer can write 4GL functions, (defined with the FUNCTION statement), a MAIN program block (defined with the MAIN statement), and reports (defined with the REPORT statement). Collectively these types of functions are often called 4GL program blocks. See also *4GL function, function, built-in function, built-in operator, MAIN program block, program block, report.*

Programmer's Environment The interface to the 4GL application development package. The Programmer's Environment is an integrated development environment that allows you to create, compile, link, run, and debug a 4GL program. See also *C Compiler Version, compile, debug, development environment, execute, link, program design database, Rapid Development System, target.*

query A request to the database to retrieve data that meets certain criteria. The SELECT statement performs database queries. In 4GL, the CONSTRUCT statement allows you to implement a "query by example." See *database, exception, output file, query by example.*

query by example A formalized way of implementing a query. The CONSTRUCT statement allows the user to enter query criteria on a screen form and creates a Boolean expression based on these criteria. This Boolean expression can then be appended to an SQL statement (usually a SELECT) to retrieve the

desired rows from the database. The SQL statement must then be prepared and executed. See also *Boolean expression, cursor, query, prepared statement, query criteria, screen form.*

query criteria A set of data values which specify qualifications to apply when looking for data to be returned in a query. The CONSTRUCT statement accepts query criteria on a screen form. See also *data entry, query by example, screen form, user interaction statement.*

Quit key The logical key that the user can press within a 4GL application to indicate cancellation of the entered data or query criteria. Pressing it requests abnormal completion of the INPUT, CONSTRUCT, PROMPT, INPUT ARRAY, or DISPLAY ARRAY statements. The physical key for Quit is CONTROL-\. If the 4GL program does not include the DEFER QUIT statement, pressing this key terminates program execution. With DEFER QUIT, pressing the Quit key sets the built-in **quit_flag** variable to TRUE but does *not* cancel the current interaction statement. See also *Accept key, exception handling, Interrupt key, logical key.*

quit_flag variable See *Quit key.*

quoted string A string enclosed in double quotation marks (“ ”). With the exception of fill characters, the contents of quoted strings are literals. See also *character, fill character, literal, string.*

Rapid Development System (RDS) One of two implementations of the 4GL application development language for UNIX systems. The **RDS** compiler produces p-code that can then be executed by a “runner.” The other implementation of 4GL for UNIX systems is the C Compiler Version; it uses preprocessors to generate C code, which is then compiled and linked to make a stand-alone, executable file. See also *C Compiler Version, compile, execute, interpret, link, p-code, preprocessor, Programmer’s Environment, program design database.*

record 1) A data structure having a fixed number of components. Each component is called a member. Members can have the same or different data types. See also *input record, record member, screen record.*

2) In uppercase, RECORD is the keyword for defining a program record in 4GL. The RECORD data type is a structured data type. In 4GL, all members of the record are accessed by listing the record name followed by the member name, with a period (.) separating them. See also *asterisk notation, program record, structured data type.*

3) In some database terminologies, a term used for a “row.” See also *row.*

record member A named component of a program record. A member can be of any 4GL data type, including RECORD or ARRAY. Some 4GL statements support the asterisk notation (*record.**) to specify all the members of a record. See also ***asterisk notation, program record, record.***

regular expression A pattern used to match variable text. The elements of a regular expression include: literal characters that must match exactly; the "wildcard" symbols—"*" to mean "one or more characters here" and "?" to mean "any one character"; and the class—a list of characters (within brackets) that are acceptable. The MATCHES and LIKE operators of SQL allow you to search for character strings that match to regular expression patterns. See also ***expression, literal, wildcard.***

relation See *table.*

relational database See *database.*

relational operators Operators that perform comparison operations. These operators return the values TRUE (=1), FALSE (=0), and in some cases UNKNOWN. (If an operand evaluates to NULL, Boolean operators can yield a third "unknown" result that 4GL treats as FALSE.) 4GL relational operators are: equal (=), not equal (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=). All relational operators have the same precedence level. See also ***associativity, binary operator, Boolean operators, operator, precedence.***

report A 4GL program block defined with the REPORT statement. A report formats data, sent as input records. The report header follows the REPORT keyword and defines the name and formal argument list (the input record) for the report. The report body (all statements between the report header and the END REPORT keywords) defines the actions of the report. See also ***argument, control block, function, input record, page header, page trailer, programmer-defined function, output file, program block.***

reserved lines Areas in a 4GL window which are set aside for use by the form or window. These areas include: Error line, where the output of the ERROR statement displays (default is the last line on the screen); Comment line, where the text in the COMMENTS field attribute displays (default is the next to last line on the screen and the last line on all other 4GL windows); Form line, where the first line of the screen form displays (default is the 3rd line of the current 4GL window); Menu line, where the ring menu displays (default is the 1st line of the current 4GL window); Message line, where the output of the MESSAGE statement displays (default is the 2nd line of the current 4GL window; and Prompt line, where the output of the PROMPT statement displays (default is the 1st line of the current 4GL window). These default

positions can be changed with the OPTIONS statement or with the ATTRIBUTES clause of the OPEN WINDOW statement. See also *4GL window, ring menu, screen, screen form.*

reserved word A sequence of letters which you cannot use in any other context of the language or program. In 4GL releases before 4.1, all keywords were reserved words. In release 4.1 and beyond, you can use keywords as variable or identifier names as long as they do not create an ambiguity for the 4GL compiler. If you compile a 4GL program as ANSI-compliant, most keywords are still reserved words and therefore should not be used as variables or identifiers. See also *ANSI-compliant, identifier, keyword.*

resources 1) The hardware and software needs of an executing program. Examples include CPU, memory, disk, printer, terminal (or workstation). These are allocated to the program's process by the operating system. See also *operating system, process, terminal, workstation.*

2) Visual and other attributes that can be chosen at run time. Resources can be chosen using command-line options or using other platform-specific methods. On X/Motif systems, resources can be specified using predefined resource names in various files. 4GL statements and ATTRIBUTE settings can override resource choices made through a platform's native resource management systems. See also *attribute.*

RETURN key The key to indicate the end-of-line. The RETURN key is the default Accept key in 4GL. See also *key.*

return value The value returned by a 4GL function to the calling routine. To return a value, the FUNCTION program block must include the RETURN statement. The calling routine must have some way of handling the function's return value(s). Reports cannot return a value to a calling function. See also *4GL function, calling routine, function definition, programmer-defined function.*

ring menu A menu in which the menu items appear in a single, horizontal line. Each menu item is called a menu option. The user can press the Spacebar or Right Arrow key to make the menu cursor traverse the menu like a "ring," as if the first option followed the last. In 4GL, the MENU statement creates a ring menu. The "highlighted" menu option is the current option. If the menu includes more options than the current 4GL window can display on a single line, the menu continues onto successive "pages," with the first page following the last. The line below the Menu line can display text that describes the current option. See also *activation key, menu, menu option, page, user interaction statement.*

roll back Terminate a transaction by undoing any changes to the database since the beginning of the transaction. The database is restored to the state that existed before the transaction began. When the transaction is rolled back, all open database cursors (except hold cursors) are closed and all locks are released. The ROLLBACK WORK statement rolls back the current transaction. See also *commit, cursor, log, transaction.*

row 1) In a database, a row is a set of related values, called columns, stored together in a table. A table holds a collection of rows, each one distinct from the others in the contents of its key. In other database terminologies, a row is sometimes called a "record" or a "tuple." See also *column, current, cursor, key, rowid, table.*

2) In a screen form, a row is the visible display of the values from one database row. The row (of data fields on the screen) may or may not be identical to a row (of values in a table in the database). A single line of a screen array is sometimes called a row. See also *screen array.*

3) In a report, a row is the information sent by the report driver function. A 4GL program generates a report by sending rows of data to a report function. These rows may or may not correspond to database rows. These rows are called "input records." See also *input record.*

4) On a screen, a row is the y-coordinate of a particular position. The x-coordinate is called a column. Several 4GL statements use rows and columns in this sense to identify location of display. See also *column, screen.*

rowid A hidden, automatically-generated column in each table of an Informix database. It uniquely identifies a row based on its position within the table. A rowid number is assigned when each row is added to a table and released when a row is deleted. Once assigned, the rowid for a particular row cannot be changed and the rowid number cannot be reused for that particular table. See also *column, row, table.*

run See *execute, interpret.*

scale The number of digits to the right of the decimal point in the representation of a number or in a data type specification. The number 3.14 has a scale of two. See also *fixed-point number, floating-point number, precision.*

scope of reference The portion of the 4GL source code in which the compiler can recognize an identifier name. The scope of reference (often referred to simply as "scope") refers to the program blocks in which an identifier can be referenced. Outside its scope, an identifier may not be defined or may even be defined differently. In 4GL, there are three levels of scope: local (a single program block), module (all program blocks in a single source module),

and global (all program blocks within a program). See also *define, global variable, identifier, local variable, module variable, name space, program block, scope, source module, variable.*

screen 1) On a character terminal, the rectangular area on a CRT in which text is displayed.The screen takes up the entire terminal display and it can display the output of only one program at a time. See also *terminal.*

2) On a workstation, the entire display in which text and, possibly, graphical objects are visible. Under a window manager in a graphical environment, a "physical screen" may contain multiple graphical windows. See also *workstation.*

3) In a 4GL application, the default 4GL window displaying in the 4GL screen. This "logical screen" is a data structure kept in memory that is a representation of a 4GL screen. The "logical screen" is not directly affected by window manager operations, though its graphical image on the "physical screen" may change. See also *4GL screen, 4GL window, column, row.*

screen array In a 4GL form, a screen array consists of consecutive lines containing identical fields and field tags. Each line of the screen array is a screen record. The screen array defines the region of the form which will display program array values. The DISPLAY ARRAY and INPUT ARRAY statements can manipulate program array values or records within a screen array. See also *field, field tag, program array, row, screen record, scrolling.*

screen field See *field.*

screen form A data-entry form displayed in a 4GL window (or the 4GL screen) and used to support input or output tasks in a 4GL application. A screen form is defined in a form specification file. Before a 4GL program can use a screen form, this file must first be compiled. The form in the current 4GL window is called the "current form." Most user interaction statements use a screen form for their input and output. See also *4GL window, 4GL screen, active form, attribute, current, form specification file, reserved lines, user interaction statement.*

screen record A named group of fields on a screen form. Screen forms have one default screen record for each table referred to in the TABLES section, including FORMONLY. The name of a default screen record is the same as the name of the table. See also *program record, record, screen array, table.*

scrolling To move forward and back (or up and down) through a series of items. Referring to a screen array, scrolling is the action of bringing invisible lines into view. Displayed data can be scrolled either vertically (to bring different rows into view) or horizontally (to show different columns). Referring

to database cursors, a sequential cursor can return only the current row and cannot return to it; but a scroll cursor can fetch any row in the active set. Thus a scrolling cursor can be used to implement a scrolling screen display. See also *cursor, screen array.*

search path The list of directories in which the operating system or a program will look for needed files. This path can be set by the user. Often, the user can specify several different paths to be searched; if one path does not lead to the file, one of the others may. For executable files, the setting of an environment variable called PATH is used. For Informix database files, the setting of the DBPATH environment variable is used. See also ***database, environment variable, operating system.***

shell A process that handles the user interaction with the operating system. From the shell, the user can execute operating system commands. A shell is usually provided to contain activity in a particular part of the computer system. In UNIX, for example, the shell handles command-line input, and standard output and error reporting. UNIX shells have their own special commands that are not usable within applications. They even have their own special variables and scripting facilities that make the user interface customizable. See also ***command line, environment variable, operating system, process.***

simple data type Any 4GL or SQL data type which has no component values. Simple data types include: integer—SMALLINT, INTEGER; floating-point—FLOAT, SMALLFLOAT, DECIMAL(***p***); fixed-point—DECIMAL(***p,s***), MONEY; time—DATE, DATETIME, INTERVAL; character—CHAR, and VARCHAR. Although individual characters in a string can be accessed, the data types CHAR and VARCHAR are considered simple data types, not structured data types. See also ***blob, character, data type, fixed-point number, floating-point number, integer, interval, structured data type.***

singleton transaction A transaction that is made up of a single SQL statement. The transaction automatically begins before each SQL statement which alters the database executes and ends when this statement completes. If the single SQL statement fails, the transaction is rolled back; otherwise it is committed. A database that is not ANSI-compliant and which does not use transaction logging uses singleton transactions. See also ***ANSI-compliant, commit, roll back, transaction.***

source file A file containing source code for a language; it is used as input to a compiler or interpreter. See also ***compile, file, interpret, source module.***

source module A module containing one or more related 4GL program blocks. A source module is a single ASCII file with the **.4gl** extension. Several source modules can be compiled and linked to produce a single executable file. See also *compile*, *executable file*, *execute*, *file*, *file extension*, *link*, *module*, *program block*.

SQL Acronym for Structured Query Language. A database query language developed by IBM and standardized by an ANSI standards committee. Informix relational database management products are based on an extended implementation of ANSI-standard SQL. See also *cursor*, *database*, *prepared statement*, *statement identifier*.

SQLCA record Acronym for SQL Communications Area. It is a built-in record that stores information about the most recently executed SQL statement. The SQLCODE member stores the result code returned by the database engine; it is used for error handling by 4GL and the Informix embedded-language products. The SQLAWARN member is a string of eight characters whose individual characters signal warning conditions. SQLERRD is an array of six integers that returns information about the results of an SQL statement. See also *database engine*, *error handling*, *status variable*.

stack A data structure that stores information linearly with all operations performed at one end (the "top"). Such types of data structures are often called LIFO (last-in, first-out) structures. Stack operations include "push," which adds a new piece of data to the top of the stack, and "pop," which removes the piece of information at the top of the stack. 4GL uses one stack to transfer arguments to C functions and another to keep track of open 4GL windows. See also *4GL window*, *call stack*.

statement An instruction that 4GL executes. This instruction is a single executable unit of program code but may cover several lines within the source module. For example, the FOR statement may have several lines between the line introduced with the FOR keyword and the line introduced with END FOR keywords. However, the FOR statement is a single statement because it performs a single action. During program execution, the statement currently being executed is often called the current statement. A statement is distinct from a command: the LET or PRINT *command* is executed by the INFORMIX-4GL Interactive Debugger; the LET or PRINT *statement* can be compiled and executed by 4GL. See also *current*, *source module*, *statement block*.

statement block A group of statements executed together. For example, all statements between the WHILE keyword and the END WHILE keywords constitute a statement block. All the statements within the AFTER INPUT block of the INPUT (or INPUT ARRAY statement) are also considered a statement block. See also *control block, program block, statement.*

statement identifier The name that represents a prepared statement created by a PREPARE statement. It is used in the management of dynamic SQL statements by 4GL and the Informix embedded language products. See also *identifier, prepared statement.*

status variable The built-in variable which 4GL sets after executing each SQL and form-related statement. If the statement is successful, status is set to zero. If the value of **status** is negative, 4GL terminates program execution unless the program contains the appropriate error handling. After execution of SQL statements, 4GL copies the value of SQLCA.SQLCODE into **status**. See also *error handling, SQLCA record.*

string A value that consists of one or more characters. You can store strings in CHAR, VARCHAR, or TEXT variables. Strings can include printable or unprintable characters, but 4GL does not provide facilities to display unprintable characters. Literal string values in 4GL statements generally must be enclosed between quotation (“ ”) marks. See also *character, literal, printable character, quoted string, subscript, substring.*

string operators Operators that perform operations on character strings. 4GL string operators are: concatenation (,) and subscripting ([ ]). Precedence for string operators is: (highest) subscripting; (lowest) concatenation. In addition, the 4GL built-in operator CLIPPED performs truncation of trailing spaces. See also *associativity, built-in operator, clipped, concatenate, operator, precedence, subscript.*

structured data type Any 4GL data type that contains component values. Structured data types include ARRAY and RECORD. Although individual characters in a string can be accessed, the data types CHAR and VARCHAR are considered simple data types, not structured data types. See also *array, data type, record, simple data type.*

subscript An integer value to access a single part or element of certain data structures like strings and arrays. In 4GL, the subscript operator is an integer value, surrounded by brackets ([ ]). For example, the syntax “**strng[3]**” accesses the third character of the CHAR string variable by specifying a subscript (or index) of 3; the syntax “**pa_customer[5]**” accesses the fifth element of the **pa_customer** program array. Two subscripts allow you to

specify the starting and ending characters. For example, "**strng[3,10]**" accesses the third through tenth characters of **strng**. See also *array, character, program array, string, string operators, substring.*

substring Consecutive characters within a string. To access a substring in a character expression, put square brackets around a pair of comma-separated unsigned integers to specify the location of the substring within a character string. For example, "**strng[3,10]**" accesses the third through tenth characters of **strng**. See also *character, subscript, string.*

system catalog Database tables that contain information about the database itself, such as the names of tables or columns in the database, the number of rows in a table, information about indexes and database privileges, and so forth. See also *column, database, index, table.*

table A collection of related database rows. It can be thought of as a rectangular array of data in which each row describes a set of related information and each column contains one piece of the information. A table sometimes is referred to as a "file" or a "relation." See also *column, cursor, database, key, row, rowid.*

target The intended result of a build. The Programmer's Environment can create executable files from multiple source modules. This process is called a build. See also *executable file, Programmer's Environment, source module.*

termcap An ASCII file in UNIX systems that contains the names and capabilities of all terminals known to the system. See also *terminal, terminfo.*

terminal A peripheral device usually centered around a raster screen. Terminals usually come with keyboards, and are used by the user of a computer system to communicate with the computer by typing commands in and looking at the output on the screen. Terminals are often character-based and are thereby distinguishable from displays which are usually graphical. Terminals support monochrome or color output, depending on their designed capabilities and computer system configuration. See also *character, key, screen, workstation.*

terminfo A database in UNIX systems that contains compiled files of terminal capabilities for all terminals known to the system. See also *terminal, termcap.*

text 1) In the SCREEN section a **4GL** form, any characters outside the fields, such as labels, titles, or ornamental lines. See also *label.*

2) In uppercase letters, TEXT is the **4GL** and SQL data type which can store up to 2^{31} bytes of character data. See also *blob.*

text cursor Pointer within a text field showing the position where typed text will be entered. Often referred to simply as the "cursor." See also *cursor, text.*

text editor System software used to create and to modify ASCII files. Usually source code is entered into a source file in a text editor. See also *ASCII, source file.*

text field A visual object for displaying, entering, and modifying text, a single line of character data. For example, form fields are text fields for use in screen forms. Text fields are used more generally, for example, to accept text in PROMPT statements. See also *field, screen form, text.*

transaction A collection of one or more SQL statements that must be treated as a single unit of work. The SQL statements within the transaction must all be successful for the transaction to succeed. If one of the statements in a transaction fails, the entire transaction can be rolled back (cancelled). If the transaction is successful, the work is committed and all changes to the database from the transaction are accepted. The transaction log contains the changes made to the database during a transaction. If a database is not ANSI-compliant, it uses singleton transactions if it does not use a transaction log and it uses explicit transactions otherwise. If a database is ANSI-compliant, it uses implicit transactions. See also *ANSI-compliant, commit, explicit transaction, implicit transaction, log, roll back, singleton transaction.*

truncation The process of discarding trailing characters from a string value, or discarding trailing digits from a number. Truncation can produce a warning or error in data type conversion, if the receiving data type has a smaller length or scale than the source data type. It can also cause rounding errors. See also *data type conversion, error, scale.*

tuple See *row.*

unary operator An operator that requires only one operand. The unary operator appears before the operand. In an expression, unary operators always have higher precedence than binary operators. In 4GL, examples include logical NOT, unary plus (+), and unary minus (-). 4GL associates most unary operators from right-to-left. See also *arithmetic operators, associativity, binary operator, Boolean operators, operand, operator, precedence.*

user An individual who interacts with a program. The user is a person who is using an application program for its intended purpose. Also referred to as the "end user." See also *application program, developer.*

user interaction statement A 4GL statement that allows a user to interact with a screen form or a field. These statements include: CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, MENU, and PROMPT. They suspend execution of the 4GL application for user input. See also *attribute, control block, data entry, query by example, ring menu, statement.*

user interface The point in a software or hardware system at which communication to or from a human can occur. In software systems, it is that part of a program that waits for input from the user of that program and displays output based on that user's input. Typical user interfaces include menus, prompts, screen forms, and on-line help messages. See also *shell.*

user locale The set of X/Open-defined environment variable settings currently active in a user session. See also *database locale, Native Language Support.*

user name A short name that identifies a particular user to a computer. It is common for it to be based on your actual name, but this is by no means the rule. Once created, the user name is used thereafter during the login process, and with establishing file ownership. See also *login, owner.*

variable A named storage location that holds a value of a specified data type. A program can access and change this value by specifying its name. A 4GL variable (sometimes called a "program variable") can transfer information between a 4GL form, report, and program. To use a 4GL variable, you must first define it (with the DEFINE statement) to specify the variable's name and data type. Variable names must follow identifier naming rules. See also *assign, constant, declare, define, global variable, identifier, local variable, module variable, name space, screen form.*

warning An exception which indicates an unexpected or abnormal condition that could lead to an error in processing or data storage. Warnings can be generated because of language syntax being used, when compiling source code or, because of a variety of processing or data exceptions when running a program. At run time, warnings can be generated by the program itself or by the database engine. By default, 4GL continues execution when it encounters a warning. The developer can change this default behavior with the WHENEVER WARNING statement. See also *database engine, exception.*

wildcard In a pattern-matching string, a character that means "any character(s) at this point." Depending on the context, a wildcard can stand for just one character (?), or it can mean "any number of any characters" (*).
For example, in the pattern "v*.4gl" the asterisk means "any number of any characters after the 'v' and preceding the period." See also *regular expression.*

workstation A peripheral device used to communicate with a computer. It usually consists of a graphical or character-based (non-graphical) display or terminal, a keyboard, serial and parallel ports for connecting to the computer and other peripherals, and a graphical input device like a mouse or trackball. See also *key, keyboard focus, screen, terminal.*

X-Open defined A classification describing environment variables which originated in the X-Open Portability Guide version 3 (XPG3) standard. This includes LANG and the LC_ variables. Collectively the values of the X-Open variables define the locale. X-Open variables rely on facilities provided by the computer manufacturer, and can vary from system to system in syntax and meaning. X-Open defined variables are distinguished from Informix-defined environment variables, which are consistent across platforms. See also *Native Language Support.*

XPG3 The X-Open Portability Guide version 3. The current standard for NLS on UNIX systems. See also *Native Language Support.*

Index

A

a command-line option 1-60, 1-62
A symbol in format strings 5-48
Abnormal termination 3-99
Accelerator keys 3-139, 3-164
Accept key
  with CONSTRUCT 3-52
  with DISPLAY ARRAY 3-92
  with INPUT 3-128
  with INPUT ARRAY 3-152
ACCEPT keyword in OPTIONS statement 3-230
Access privileges
  checking 3-196, 4-90
  database 3-59
  with ASCII files 3-181, 3-274
Accounting parentheses 4-92
ACTION Menu (upscol utility) B-6
Activation clause
  CASE statement 3-22
  CONSTRUCT statement 3-38
  DISPLAY ARRAY statement 3-87
  IF statement 3-124
  INPUT ARRAY statement 3-156
  INPUT statement 3-134
  MENU statement 3-195
  PROMPT statement 3-258
  WHILE statement 3-287
Activation key
  CONSTRUCT control block 3-41
  DISPLAY ARRAY control block 3-87
  INPUT ARRAY control block 3-161
  INPUT control block 3-137
  MENU control block 3-199
  PROMPT control block 3-258
Active set 3-107, 3-283
Addition (+) operator
  number expressions 3-339, 4-22
  precedence 3-328
  precision and scale 3-321
  reserved lines 3-226, 3-231
  returned values 4-20
  time expressions 3-357, 4-22
AFTER CONSTRUCT block 3-43
AFTER DELETE block in INPUT ARRAY statement 3-165
AFTER FIELD block
  CONSTRUCT statement 3-42, 4-68
  INPUT ARRAY statement 3-164
  INPUT statement 3-139
AFTER GROUP OF block 3-261, 4-95, 6-29
AFTER INPUT block
  INPUT ARRAY statement 3-166
  INPUT statement 3-140
AFTER INSERT block, INPUT ARRAY statement 3-165
AFTER keyword
  CONSTRUCT statement 3-42, 3-43
  INPUT ARRAY statement 3-165
  INPUT statement 3-134
  REPORT statement 3-261, 6-29
AFTER ROW block, INPUT ARRAY statement 3-166, 4-26
Aggregate function
  AVG( ) 4-14, 6-46
  COUNT(*) 4-14, 6-47, 6-49
  GROUP 4-14, 6-31, 6-46
  MAX( ) 4-14, 6-47

- MIN( ) 4-14, 6-47
- PERCENT(*) 3-261, 4-14, 6-47
- SUM( ) 4-14, 6-46
- two-pass reports 3-262
- view columns 3-72, 3-314, 5-29
- with blob arguments 3-298, 3-318
- with NULL values 4-13, 6-47
- with reports 3-100, 3-262, 6-31
- with SQL statements 6-45

Alias of a table
- CONSTRUCT statement 3-36
- in a field clause 3-359
- in a form 3-361, 5-19, 5-29

ALL keyword
- MENU statement 3-194
- SQL Boolean operator 3-330

Allocation of resources 3-66, 3-186

ALTER INDEX statement, interrupting 3-236

ALTER TABLE statement
- impact of NLS on E-10
- interrupting 3-236
- query by example 3-35, 3-363

Ambiguous selections in menus 2-16

Ampersand (&) symbol 4-92

AND operator
- Boolean operator 3-32, 3-329, 3-333, 4-31, 4-37
- precedence of operators 3-328, 4-37
- with BETWEEN 4-35, 5-33, 5-71

Angle (< >) brackets 3-48, 3-298, 3-334, 4-32, 5-53, 6-44

ANSI E-5, E-15, E-35, E-39, E-48

ansi
- option of c4gl command 1-31
- option of fglpc command 1-59

ANSI-compliance
- and DBANSIWARN D-8
- -ansi flag D-8

ANSI-compliant database
- comment indicators 2-6
- database references 3-361, 3-362
- default attributes 5-72
- default values 3-127
- error handling 3-283, 3-285, 4-85
- initializing variables 3-127
- interrupting transactions 3-237, 3-238, 3-239
- opening 3-61
- owner naming 3-35, 3-69, 3-126, 3-279, 3-361, 5-19
- remote 3-362
- validation criteria 3-280

ANY keyword
- SQL Boolean operator 3-330
- WHENEVER statement 2-26, 3-281

ANYERR 3-283, 3-319, 4-50

Application
- internationalizing I-3
- programming interface to C 4-6, C-1
- program, compiling 1-24, 1-29, 1-58

Argument
- for 4GL program command line 4-17, 4-76
- in function calls 3-17, 3-281
- in report definition 3-261, 6-6
- passed to a C function C-23
- passing by reference 3-18, 3-190, 3-242, 3-264, 4-8
- passing by value 3-18, 3-242, 3-263, 4-8
- stack 2-27, C-2, C-12

ARG_VAL( ) 4-16

Arithmetic functions C-36

Arithmetic operators
- binary 3-339, 4-19
- integer expressions 3-339
- number expressions 3-341, 4-18
- precedence of operators 3-328
- time expressions 3-357, 4-18, 4-21, 4-54
- unary 3-340, 4-19

Array
- of records 4-26
- program array 4-24
- screen array 5-63, 5-66

ARRAY data type
- declaration 3-71, 3-297
- in DISPLAY ARRAY statements 3-85
- in FOREACH statement 3-107
- in MENU statement 3-206
- in report parameter list 3-261, 6-8
- index 3-329

ARRAY keyword
- DEFINE statement 3-71
- DISPLAY ARRAY statement 3-85
- INPUT ARRAY statement 3-152

Arrow keys
- CONSTRUCT statement 3-51
- INPUT ARRAY statement 3-172
- INPUT statement 3-146
- MENU statement 3-209
- termcap entry F-5
- WORDWRAP fields 3-148, 5-61

Arrows in syntax diagrams Intro-7

ARR_COUNT( )
- syntax and description 4-24
- with DISPLAY ARRAY 3-85, 3-90
- with INPUT ARRAY 3-171

ARR_CURR( )
- syntax and description 4-26
- with DISPLAY ARRAY 3-90
- with INPUT ARRAY 3-171

ASCII character set E-2, E-3, E-5, E-8, E-23, E-30

ASCII characters
- and corresponding codes G-2
- ASCII operator 3-77, 3-214
- collating sequence 3-49, 3-344, 5-45, G-2
- from integer codes 4-28
- in screen layouts 5-16
- printable 3-345
- unprintable 3-345

ASCII collation E-4, E-17, E-22

ASCII file
- colornames F-19
- data input 3-182
- data output 3-274
- error log 4-51, 4-84
- form specification 5-3
- Help messages 2-22
- source code module 2-7
- .4gl source files 1-24, 1-53

ASCII operator
- precedence of operators 3-328
- PRINT statement 4-29, 6-47

Asian Language Support I-2

Assignment statements 2-7, 3-125, 3-178, 5-20

Associativity of operators 3-328
Asterisk (*) notation
  arithmetic operator 4-64
  database columns 3-33, 3-363
  exponentiation operator 3-328, 3-339, 4-20
  in REPORT prototype 3-261, 6-6
  multiplication operator 3-328, 3-339, 3-357, 4-20, 4-22
  overflow in data conversion C-26
  program record members 3-314, 3-363
  screen array elements 3-82, 3-363
  screen field overflow 3-80, 3-86, 4-91, 5-14
  screen record members 3-359, 3-363, 4-66, 5-65
  wildcard with CONSTRUCT 3-49
  wildcard with MATCHES 3-335, 4-34
  with COUNT function 4-14, 6-47
  with PERCENT function 3-261, 4-14, 6-47
Asynchronous message handling 3-62
AT keyword
  DISPLAY statement 3-74
  OPEN WINDOW statement 3-220
At (@) symbol
  database servers Intro-10, 3-33, 3-58
  MENU statement 3-199, 4-60
  table or column prefix 2-12, 5-4
ATTRIBUTE keyword
  CONSTRUCT statement 3-37, 5-71
  DISPLAY ARRAY statement 3-86, 5-71
  DISPLAY FORM statement 3-93, 5-72
  DISPLAY statement 3-83, 5-71
  ERROR statement 3-97
  INPUT ARRAY statement 3-155, 5-71
  INPUT statement 3-133, 5-71
  MESSAGE statement 3-214
  OPEN WINDOW statement 5-72
  OPTIONS statement 3-230, 5-72
  PROMPT statement 3-257
Attribute types
  AUTONEXT 5-28, 5-30, 5-70
  BLACK 3-290
  BLINK 3-291, 5-31, 5-71, F-10, F-29
  BLUE 3-291, 5-34
  BOLD 3-291, 5-34, 5-71, F-29
  BORDER 3-224, F-6, F-25
  COLOR 3-298, 3-317, 3-331, 5-28, 5-31, 5-78, B-8
  COMMENTS 5-28, 5-36, 5-70
  CYAN 3-290
  DEFAULT 5-25, 5-28, 5-38, 5-70, B-7
  DIM 3-291, 5-34, 5-71, F-29
  DISPLAY LIKE 5-18, 5-28, 5-40
  DOWNSHIFT 5-28, 5-41, B-7
  FORM 3-234
  FORMAT 3-81, 5-28, 5-42, B-9
  GREEN 3-290
  INCLUDE 3-279, 5-25, 5-28, 5-44, 5-70, B-7
  INVISIBLE 3-83, 3-93, 3-214, 3-291, 5-28, 5-46, 5-71, F-29
  LEFT 3-81, 5-31, B-9
  MAGENTA 3-290
  NOENTRY 5-28, 5-47
  NORMAL 3-214, 3-291, 5-34, 5-71
  PICTURE 5-48, 5-70
  PROGRAM 3-81, 3-298, 3-317, 5-28, 5-50
  RED 3-291, 5-34
  REQUIRED 5-28, 5-52
  REVERSE 3-224, 3-291, 5-28, 5-31, 5-53, 5-71, F-4, F-10, F-29
  SHIFT 5-70, B-7
  UNDERLINE 3-291, 5-31, 5-71, F-10, F-29
  UPSHIFT 5-28, 5-54, B-7
  VALIDATE LIKE 5-18, 5-28, 5-55
  VERIFY 5-28, 5-56, 5-70
  WHITE 3-291, 5-34
  WORDWRAP 5-26, 5-28, 5-57
  YELLOW 3-290
ATTRIBUTES section of form specification
  default values 5-38, 5-69, B-7
  field attributes 5-20, 5-28, 5-31
  field names 5-20, 5-28
  field tags 5-21, 5-33
  fields linked to columns 5-18, 5-22
  FORMONLY fields 5-20, 5-24
  multiple-segment fields 5-26, 5-57
  multiple-table forms 5-9
  syntax 5-20, 5-28
AUTONEXT attribute 5-28, 5-30, 5-70, B-7
AVG( ) aggregate function 4-14, 6-46

B

Background process 3-267
Backslash (\ ) symbol
  default Quit key 3-62
  escape character 3-178
  in forms 5-16
  in input files 3-184
  in output files 3-276
  in pathnames 3-59
  with LIKE 3-336, 4-34
  with MATCHES 3-335, 4-34
Backspace key for menus 2-16, 3-207
BEFORE DELETE block in INPUT ARRAY statement 3-160
BEFORE FIELD block
  CONSTRUCT statement 3-40
  INPUT ARRAY statement 3-161
  INPUT statement 3-137
BEFORE GROUP OF block
  definition of 6-31
  variables 3-261
BEFORE INPUT block
  INPUT ARRAY statement 3-159
  INPUT statement 3-136
BEFORE INSERT block in INPUT ARRAY statement 3-160
BEFORE keyword
  CONSTRUCT statement 3-39, 4-65
  INPUT ARRAY statement 3-160, 3-161, 4-65
  INPUT statement 3-137, 4-65

MENU statement 3-196
REPORT statement 3-261, 6-31
BEFORE MENU block 3-196
BEFORE ROW block in INPUT ARRAY statement 3-160, 4-26
BEGIN WORK statement 3-237
Bell, ringing 3-96, 3-179, 4-28, 5-48
BETWEEN operator 3-330, 4-35, 5-33, 5-71
Binary arithmetic operators 3-329, 3-339, 4-19, 4-20
Binary large objects (blobs)
data types 3-296
defining variables 3-70
in Boolean expressions 4-33, 5-33
in screen forms 5-50
passing by reference 3-18, 3-242
Binding
of forms to database 5-21
of variables to screen fields 3-34, 3-81, 3-85, 3-130, 3-153, 5-3
BLACK attribute 3-290, 5-31, 5-46, 5-71, F-20
Blank characters
as list separators Intro-8
between menu options 3-206
CLIPPED operator 3-328, 4-38
DATETIME separator 3-302, 3-351
default character value 3-131, 3-154, 5-12, 5-25, 5-38
in identifiers 2-10
in input files 3-182
in literal numbers 3-342
in output files 3-275
in output strings 4-93
INTERVAL separator 3-309, 3-355
padding CHAR values 3-318
PICTURE attribute 5-48
SPACE or SPACES operator 4-82, 6-50
trailing blank spaces 3-318, 4-38
versus NULL values 5-44
with FORMAT attribute 5-43
within statements 2-3
WORDWRAP fields 3-147, 5-58, 5-59
WORDWRAP operator 4-103, 6-51
BLINK attribute 3-290, 3-291, 5-31, 5-71, F-10, F-29
Blob
description of 3-70
storing data in 3-298
BLUE attribute 3-290, 5-31, 5-71, F-20
BOLD attribute 3-290, 5-71, F-29
Boldface terms in text Intro-5
Boolean capabilities F-3, F-22
Boolean expression
CASE statement 3-21
CONSTRUCT statement 3-31
IF statement 3-124
in 4GL statements 3-333
in SQL statements 3-330
in syscolatt B-9
in syscolatt table 5-71
logical operators 3-334, 4-32
WHILE statement 3-287
wildcards in searches 3-336, 4-34
with COLOR attribute 5-32, 5-78
Boolean operators
AND 3-329, 3-333, 4-31, 4-35
BETWEEN 3-330, 4-35
description of 4-30
IN 3-330, 4-35
IS NOT NULL 3-328, 4-33
IS NULL 3-328, 4-33
LIKE 3-328, 4-33
MATCHES 3-328, 4-33
NOT 3-329, 3-333, 4-31
OR 3-333, 4-31
Bordered window, graphics characters used 5-17, F-25
BOTTOM MARGIN keywords 6-11, 6-38
Bourne shell
how to set environment variables D-4
.profile file D-2
Braces ({ }) symbols
comment indicator 2-6
screen layout of forms 5-14
Brackets ([ ]) symbols
in string comparisons 3-335, 4-34
records within screen arrays 3-36, 3-359, 4-68, 5-67
subsets of BYTE values 3-298
substrings in character arrays 3-297
substrings of TEXT columns 3-317
to specify program arrays 3-328, 3-344
to specify screen arrays 5-63, 5-66
to specify search criteria 3-50, 3-335
to specify substrings 3-77, 3-215, 3-328, 3-344, 5-22
with SCROLL 3-268
Built-in functions
Aggregates 4-13
ARG_VAL( ) 4-16
ARR_COUNT( ) 4-24
ARR_CURR( ) 4-26
DOWNSHIFT( ) 4-47
ERRORLOG( ) 4-51
ERR_GET( ) 4-48
ERR_PRINT( ) 4-49
ERR_QUIT( ) 4-50
FGL_DRAWBOX( ) 4-56
FGL_GETENV( ) 4-58
FGL_KEYVAL( ) 4-60
FGL_LASTKEY( ) 4-62
introduction to 4-5
LENGTH( ) 4-71
NUM_ARGS( ) 4-76
SCR_LINE( ) 4-78
SET_COUNT( ) 4-80
SHOWHELP( ) 4-81
SQLEXIT( ) 4-83
STARTLOG( ) 4-84
UPSHIFT( ) 4-90
Built-in operators
Arithmetic 4-18
ASCII 3-328, 4-28
Boolean 4-30
CLIPPED 3-328, 4-38
COLUMN 3-328, 4-40
CURRENT 3-329, 4-42, 5-39
DATE 3-329, 3-330, 4-44
DATE( ) 3-329, 4-45
DAY( ) 3-329, 4-46
EXTEND( ) 3-329, 4-53

FIELD_TOUCHED( ) 3-330, 4-64
GET_FLDBUF( ) 3-330, 4-66
INFIELD( ) 3-330, 4-69
LINENO 3-330, 4-73
Logical operators 4-31
MDY( ) 3-329, 4-74
MOD 3-328
MONTH( ) 3-329, 4-75
PAGENO 3-330, 4-77
Relational operators 4-32
SPACE or SPACES 3-328, 4-82
TIME 3-329, 3-330, 4-86
TODAY 3-329, 4-87, 5-39
UNITS 3-328, 5-39
USING 3-328, 4-91
WEEKDAY( ) 3-329, 4-100
WORDWRAP 3-328, 4-102
YEAR( ) 3-329, 4-104
Built-in SQL functions 4-5, 4-6
Built-in variables
int_flag 3-62, 3-138, 3-162, 3-199, 3-235
quit_flag 3-62, 3-138, 3-162, 3-199, 3-235
SQLAWARN 2-25, 3-61, 3-320
SQLCA record 2-23, 3-282
SQLCODE 2-24, 4-48
SQLERRD 2-24
SQLERRM 2-24
SQLERRP 2-24
status 2-23, 3-279, 3-281, 3-319, 4-48
BY keyword
CONSTRUCT statement 3-34
DISPLAY statement 3-74
Form specification file 5-12
INPUT statement 3-131, 5-4
REPORT statement 3-262, 6-18
SCROLL statement 3-268
BY NAME clause
CONSTRUCT statement 3-34
DISPLAY statement 3-74
INPUT statement 3-131, 5-4
BYTE data type
ASCII representation 3-182, 3-275
Boolean expressions 4-33, 5-33
data entry 3-150, 3-174
declaration 3-65, 3-70
description 3-298
display fields 3-81, 3-86, 5-23, 5-50
display width 5-76, 6-44
in expressions 3-332
in program records 3-72, 3-313
in report output 6-44
initializing 3-186
large data types 3-296
passing by reference 3-18, 3-264
query by example 3-48
selecting a BYTE column 3-298
syscolval table 3-280, 5-55
upscol 5-70

C

C Compiler version of 4GL 1-3, 1-6
C compiler, function 1-29
C language
API 4-6
functions 1-30, 1-62, 1-64, 3-16, 4-6, C-6
C shell
how to set environment variables D-4
.cshrc file D-2
.login file D-2
c4gl command 1-5, 1-29
CALL keyword in WHENEVER statement 3-281, 3-285
CALL statement
description 3-16
with C functions 1-68
Calling routine 3-16, 3-263, 4-8, 6-4
Caret (^) symbol
with CONSTRUCT 3-50
with MATCHES 3-335, 4-34
with TOP OF PAGE 6-17
CASE statement 3-21
cat utility 1-61
Category, NLS E-2
cfglgo command 1-64, 1-67
CHAR data type
data type conversion 3-319, 3-324, C-26
declaration 3-68, 3-294
description 3-299
display fields 5-42, 5-48, 5-57
display width 3-78, 5-76, 6-44
in expressions 3-343
in NLS E-4, E-6, E-9, E-18
in report output 6-44
returned by functions 3-19, 3-264
searching with LIKE 3-336, 4-34
searching with MATCHES 3-335, 4-34
subscripts 5-22, 5-25
unprintable characters 3-345
Char data type (of C) C-25
CHAR keyword
DEFINE statement 3-68, 3-294
PROMPT statement 3-257
Character
data types 3-296, 3-343
position 5-22
Character expression
CLIPPED operator 4-38
data type conversion 3-325
description of 3-343
NULL values 5-25
searching with LIKE 3-329, 3-336, 4-34, 4-37, 4-101
searching with MATCHES 3-329, 3-335, 4-34, 4-37
substring 3-215, 5-22
syntax 3-343
Character set
defined E-3
mentioned E-23, E-30
Character string
as Boolean expression 3-32
as DATETIME value 3-303
as INTERVAL value 3-311
determining the length 4-71
printable characters 3-345
Child process 3-265
CLEAR statement 3-26
CLIPPED operator
description of 4-38
DISPLAY statement 3-78
in a string expression 4-71
MESSAGE statement 3-213
precedence of operators 3-328
PRINT statement 6-44
CLOSE DATABASE statement 3-59, 3-101, 4-83
CLOSE FORM statement 3-29

CLOSE WINDOW statement 3-30
Collation E-2, E-4, E-5, E-8, E-10, E-13, E-20, E-27
COLLCHAR environment variable
  interaction with DBNLS E-16, E-19
  mentioned E-2, E-4, E-5, E-11
  settings E-19
  syntax E-18
Colon (:) symbol
  after database name Intro-10, 3-361
  after label identifiers 3-177
  after menu name 3-208
  before label identifier 3-123
  DATETIME separator 3-49, 3-302, 3-351, 4-86
  field specification separator F-4
  INTERVAL separator 3-49, 3-309, 3-311, 3-355
  ranges with CONSTRUCT 3-49
Color
  number codes 5-70
  screen displays 3-223, 3-291, 5-27, 5-34
  setting INFORMIXTERM for D-35
COLOR attribute 5-28, 5-31, 5-78, F-20
colornames file B-8, F-10, F-19
Column
  changing data type 3-35, 3-319, 3-363
  in screen arrays 5-66
  inserting data 3-181, 3-184
  upscol utility 5-69
  with LIKE 3-125, 3-278
  with table qualifier 3-361
COLUMN keyword
  COLUMN operator 3-328, 4-40, 6-48
  DISPLAY statement 3-77
  MESSAGE statement 3-214
  PRINT statement 3-254
COLUMN operator 4-40
Columns
  in stores database A-3 to A-7
  in stores database tables A-3
  in upscol tables B-5
  locale-sorted E-6, E-8
COLUMNS keyword in OPEN WINDOW statement 3-221
Comma (,) symbol
  array subscripts 3-297
  in literal numbers 3-340, 3-342
  in substring specifications 3-215
  in USING format strings 4-92
  separator in lists 3-364, F-22
COMMAND keyword, MENU statement 3-197, 3-200
Command line
  arguments of a 4GL program 4-17, 4-76
  RUN statement 3-265
  START REPORT statement 3-273
  to compile a message file B-3
  to compile a screen form 5-75
  to create a customized runner 1-67, 1-70
  to invoke a 4GL program 1-5, 1-29, 1-62, 1-63, 1-70, 4-17, 4-76
  to invoke compiler 1-5, 1-30, 1-59
Comment indicators 2-6, 5-13, 5-16, F-3, F-22
COMMENT keyword
  OPEN WINDOW statement 3-223
  OPTIONS statement 3-229
Comment line 2-19, 3-51, 3-94, 3-172, 3-226, B-7
COMMENT LINE keywords
  OPEN WINDOW statement 3-223
  OPTIONS statement 3-229
COMMENTS attribute 5-36, 5-70, B-7
COMMIT WORK statement
  interrupting transactions 3-237, 3-239
  with LOAD 3-185
Comparison operators 3-48, 3-329, 3-334, 3-337, 3-358, 4-32, 4-36, 4-37
Compatible data types 3-324
COMPILE Menu 1-24
Compile option
  FORM Menu 1-15, 1-44, 5-74
  MODULE Menu 1-10, 1-39
  PROGRAM Menu 1-20, 1-49
Compiler
  C compiler 3-305
  directive statements 3-14
  maximum number of variables allowed 3-67
  mkmessage 4-81
  p-code 3-67
Compile-time errors 1-9, 1-38, 5-74
Compiling
  command line 1-29, 1-30, 1-58
  help messages B-2
  in Programmers Environment 1-8, 1-29, 1-37, 1-58
  programs that call C functions 1-63
  screen forms 1-13, 1-43
  with ansi flag 1-31, 1-59
Compound statements 2-5, 2-8, 3-55, 3-95
COMPRESS keyword, WORDWRAP attribute 3-147, 5-57, 5-59
Conditional statements
  CASE statement 3-21
  COLOR attribute 5-31, 5-78
  IF statement 6-37
  NEED statement 6-40
  syscolatt file B-9
  syscolatt table 5-71
CONNECT statement, and INFORMIXSERVER environment variable D-33
Connection
  setting the INFORMIXCONRETRY environment variable D-30
  setting the INFORMIXCONTIME environment variable D-31
Consistency checking
  defined E-6, E-15
  mentioned E-7, E-27, E-49
  overriding with Open NLS E-7
Constant
  Boolean 3-333
  floating-point 3-306
  integer 3-340
  operand 3-331

time interval 3-310
time-of-day values 3-302
CONSTRAINED keyword in OPTIONS statement 3-51, 3-230
CONSTRUCT keyword
AFTER CONSTRUCT block 3-43
BEFORE CONSTRUCT block 3-39
CONSTRUCT statement 3-31
CONTINUE CONSTRUCT 3-46
END CONSTRUCT statement 3-47
EXIT CONSTRUCT statement 3-47
Context of variable declarations 3-66
CONTINUE keyword
CONTINUE CONSTRUCT 3-46
CONTINUE FOR 3-103
CONTINUE FOREACH 3-108
CONTINUE INPUT statement 3-143, 3-169
CONTINUE MENU 3-202
CONTINUE WHILE 3-288
description 3-55
WHENEVER statement 3-281, 3-285, 4-51, 4-85
Control blocks
AFTER GROUP OF 6-29
BEFORE GROUP OF 6-31
CONSTRUCT statement 3-39
description 2-8
DISPLAY ARRAY statement 3-87
FIRST PAGE HEADER 6-33
IF statement 3-124
in FORMAT section of a report 6-27
INPUT ARRAY statement 3-156, 3-158
INPUT statement 3-136
MENU statement 3-55, 3-194
ON EVERY ROW 6-34
ON LAST ROW 6-36
PAGE HEADER 3-269, 6-37
PAGE TRAILER 3-269, 6-38
CONTROL keys
CONSTRUCT statement 3-41
INPUT ARRAY statement 3-161, 3-173
INPUT statement 3-137, 3-138
PICTURE attribute 5-49
WORDWRAP fields 3-149, 5-61
Conventions
syntax Intro-6
typographical Intro-5
COUNT(*) aggregate function 4-14, 6-47
CPU cost for a query 2-24
CREATE INDEX statement, interrupting 3-236
CREATE TABLE statement E-4, E-10
crtcmap utility E-23
Currency symbol
default (= $) 5-38
in format strings 4-94
in input files 3-182
in literal numbers 3-342
in output files 3-275
Current
database 3-58, 3-274
form 3-31, 3-56, 3-128, 3-152, 3-255
menu option 2-18
option of a menu 1-7, 1-35
window 2-19, 3-27, 3-31, 3-56, 3-128, 3-152
CURRENT keyword
Boolean expressions 5-31
CURRENT operator 4-42, 5-39
CURRENT WINDOW statement 3-56
CURRENT operator 4-42
CURRENT WINDOW statement 3-56
Cursor
manipulation statements 3-12
scope of reference 2-11
visual cursors 2-18
CURSOR keyword in DECLARE statement 3-32
Cursor movement
CONSTRUCT statement 3-50
defined in terminfo file F-24
DISPLAY ARRAY statement 3-91
editing keys 3-52, 3-147, 3-172, 3-173
in a screen form 5-30
in a screen record 5-21
INPUT ARRAY statement 3-154
INPUT statement 3-130
MENU statement 2-16, 3-209
NEXT FIELD clause 3-44, 3-142, 3-168
NEXT OPTION clause 3-203
unprintable characters 3-345
customer table in stores database A-3
Customized runners 1-48, 1-64
CYAN attribute 3-290, 5-31, 5-71, F-20
C++ language 2-10

D

d symbol in format strings 4-94, 5-42
Data
access statements 3-12
definition statements 3-12
entry 3-147, 3-173, 5-6, 5-46, 5-52
integrity statements 3-13
manipulation statements 3-12
Data input
INPUT ARRAY statement 3-152
INPUT statement 3-128
LOAD statement 3-181
Data types
ARRAY 3-71, 3-297, 5-20
blob 3-70, 3-242, 3-296
BYTE 3-70, 3-296, 3-298, 3-332, 4-33, 5-23, 5-40, 5-50, 5-55, 5-76, 6-44
C language C-24
CHAR 3-296, 3-299, 3-343, 4-40, 5-38, 5-57, 5-76, 6-44
CHARACTER 3-300
character 3-296
CHAR, in NLS E-4, E-6, E-9, E-18
conversion between 2-25, 3-319, 3-324, 4-20, 4-45, 4-54
DATE 3-295, 3-300, 3-348, 4-40, 4-91, 5-38, 5-76, 6-44
DATETIME 3-295, 3-300, 3-348, 4-40, 5-39, 5-76, 6-44
DATE, in NLS E-9

DEC 3-304
DECIMAL 3-304, 3-320, 3-341, 4-19, 5-42, 5-76, C-23
DECIMAL, in NLS E-9
declaration 3-65, 3-293
display width 6-44
DOUBLE PRECISION 3-305
fixed point 3-295
flat file format 3-182, 3-275
FLOAT 3-305, 3-341, 4-19, 5-42, 5-76
floating point 3-295, 3-306, 3-315
FLOAT, in NLS E-9
indirect declaration 3-60, 3-69
INT 3-306
INTEGER 3-306, 3-341, 4-19, 5-76
INTERVAL 3-295, 3-307, 3-353, 3-357, 4-19, 4-40, 5-39, 5-76, 6-44
keywords 3-293
large binary 3-70, 3-149, 3-174, 3-186, 3-296
MONEY 3-312, 3-341, 4-19, 4-40, 4-91, 5-38, 5-76, 6-44
MONEY, in NLS E-9
NCHAR E-4, E-6, E-9, E-18
number 3-295
NUMERIC 3-313
NVARCHAR E-4, E-6, E-9, E-18
REAL 3-313
RECORD 3-72, 3-154, 3-313, 5-20
SERIAL 3-69, 3-130, 3-153, 4-40, 5-47, 6-44
simple 3-68, 3-294
SMALLFLOAT 3-315, 3-341, 4-19, 5-42, 5-76
SMALLFLOAT, in NLS E-9
SMALLINT 3-316, 3-341, 4-19, 5-76
structured 3-70, 3-296
TEXT 3-70, 3-296, 3-317, 3-332, 4-33, 4-102, 5-50, 5-57, 5-76, 6-50
time 3-295
VARCHAR 3-14, 3-296, 3-318, 3-343, 5-57, 5-76
VARCHAR, in NLS E-4, E-6, E-9, E-18
whole number 3-295

Data validation
INCLUDE attribute 5-45
NOENTRY attribute 5-47
upscol utility 5-69, 5-71, B-7
VALIDATE LIKE attribute 5-55
VERIFY attribute 5-56

Database
administrator (DBA) access privileges 1-16
ANSI-compliant 3-61, 4-85, 5-18, 5-72
binding to screen forms 5-4
closing 3-59, 3-101, 4-83
creating 2-25
current 3-60, 3-274, 3-362
default 3-59, 3-127, 3-280
engine 3-58, 3-61, 3-296, 3-330, 3-361, 4-83, 5-11
exclusive mode 3-61
explicit transactions 3-237, 3-238
locale E-2
lock 3-61
map of stores A-8
naming rules 2-10
opening 3-58
remote 3-58, 3-362, 4-83
schema 5-25
server 3-58, 3-278, 3-361, 3-362, 5-11
server, specifying default for connection D-33
singleton transactions 3-237, 3-238
specification 3-59
stores demonstration A-1
stores2 Intro-12
types of transactions 3-237
with transactions 3-61, 3-108, 4-83

Database cursor
FOREACH statement 3-105
naming rules 2-10

Database name
DATABASE statement 3-58
table qualifier 3-33, 3-361

DATABASE section of form specification
creating as FORMONLY 5-11, 5-24
syntax 5-10
WITHOUT NULL INPUT 5-12, 5-38

DATABASE statement
database connection 4-83
indirect typing 3-69
syntax and description 3-58
syscolval table 3-126, 3-280
two-pass reports 3-101
with SQLEXIT( ) 4-83

DATE data type
arithmetic operations 3-356, 4-23
converting to DATETIME 3-321, 4-53
converting to other data types 3-324
declaration 3-68, 3-294
default value 5-25, 5-38
description 3-300
display fields 3-131, 3-154, 4-94, 5-38, 5-42
display width 3-78, 5-76, 6-44
formatting 4-94
in integer expressions 3-356, 4-23
in NLS E-9
in report output 6-44
in time expressions 3-348
literal values 3-182, 3-275, 3-349, 4-94
time data type 3-295
values 3-348

DATE keyword
DATE data type 3-300
DATE operator 3-330, 4-44, 5-39
DATE( ) operator 4-45
DEFINE statement 3-294

DATE operator 4-44
DATE value formatting D-9, D-18

DATETIME data type
arithmetic operations 3-356, 4-54
as character string 3-303, 3-322
data type conversion 3-321, 3-324, 4-45, 4-53
declaration 3-68, 3-294, 3-300
default value 5-25, 5-38
display fields 5-39, 5-49
display width 3-78, 5-76, 6-44
in report output 6-44
in time expressions 3-348
literal values 3-182, 3-275, 3-351

qualifiers 3-301, 3-349, 5-39
time data types 3-295
values 3-348
DATE( ) operator 4-45
DAY keyword
CURRENT operator 4-42
DATETIME qualifier 3-301, 3-349, 4-42
DAY( ) operator 4-46
EXTEND( ) operator 4-54
INTERVAL qualifier 3-309, 3-353, 4-55
UNITS operator 4-89
DAY( ) operator 4-46
DBANSIWARN environment variable 2-25, D-8
DBAPICODE environment variable E-2, E-11, E-23
DBDATE environment variable 3-32, 3-182, 3-275, 3-300, 3-325, 4-45, 4-94, D-9, E-3, E-11, E-42, I-5
DBDELIMITER environment variable 3-184, 3-276, D-11
DBEDIT environment variable 1-10, 1-24, 1-38, 1-53, D-11
dbexport utility, specifying field delimiter with DBDELIMITER D-11
DBFLTMASK environment variable 3-32
DBFORM environment variable D-12, E-3, E-11, E-24
DBFORMAT environment variable 3-32, 3-342, 4-91, D-14, E-3, E-5, E-11, E-35, E-38, E-39, E-41, E-42, I-5
DBLANG environment variable B-5, D-18, E-3, E-11, E-24, E-42, I-5
dbload utility, specifying field delimiter with DBDELIMITER D-11
DBMONEY environment variable 3-32, 3-312, 3-325, 3-342, 4-91, D-21, E-3, E-5, E-11, E-35, E-38, E-41, E-42
DBNLS environment variable
interaction with COLLCHAR E-16, E-19
mentioned E-2, E-4, E-5, E-11
settings E-17
syntax E-16
DBPATH environment variable 1-16, 1-26, 1-55, 3-33, 3-59, D-23
DBPRINT environment variable 3-272, D-26
DBREMOTECMD environment variable D-27
DBSPACETEMP environment variable D-28
DBTEMP environment variable 3-188, D-29
DBTIME environment variable 3-32
DBUPSPACE environment variable D-29
De-allocation of variables 3-190
Debug option
MODULE Menu 1-40
PROGRAM Menu 1-51
Debugger Intro-4, 1-36, 1-51, 1-56, 1-58, 1-63, 3-99
decadd( ) C-36
deccmp( ) C-37
deccopy( ) C-38
deccvasc( ) C-25
deccvdbl( ) C-34
deccvflt( ) C-32
deccvint( ) C-28
deccvlong( ) C-30
decdiv( ) C-36
dececvt( ) C-39
decfcvt( ) C-39
DECIMAL data type
arithmetic operations 4-20
data type conversion 3-319, 3-324
declaration 3-68, 3-294
description 3-304
display fields 5-43
display width 3-78, 5-76, 6-44
floating point 3-304
in NLS E-9
in report output 6-44
internal representation C-23
literal values 3-342
scale and precision 3-320
DECIMAL functions for C C-23
Decimal separator D-14, D-21, E-31, E-33, E-35, E-40, E-45
Decimal (.) point
DATETIME separator 3-302, 3-351
DECIMAL values 3-304
fixed-point values 3-304
FLOAT values 3-305
floating-point values 3-304, 3-316
in format strings 4-92
in literal numbers C-25
INTERVAL separator 3-309, 3-353
literal numbers 3-342
decimal.h file C-23
Declaration statements 2-7, 2-9
DECLARE statement
declaring a cursor 3-106
query by example 3-32
SQLCA record 2-24
decmul( ) C-36
decsub( ) C-36
dectoasc( ) C-26
dectodbl( ) C-35
dectoflt( ) C-33
dectoint( ) C-29
dectolong( ) C-31
dec_t structure C-23
Default
activation key 3-198
assumptions for your environment D-5
attributes 5-69
database 3-59, 3-126, 3-278, 3-280
editor 1-24, 1-53
error record 4-51
error-handling action 4-85
field attributes 3-292, 5-72
field label 5-76
field width 5-76
Help key 2-21
precision 4-53
report margins 4-103, 6-10, 6-51
reserved line positions 3-94, 3-226
screen layout 5-76
screen record 5-64

validation criteria 3-280
values 3-127
window attributes 3-223
DEFAULT attribute
field attribute 5-28
syntax and description 5-38
with INITIALIZE statement 5-71
with INPUT 3-130
with INPUT ARRAY 3-154
with WITHOUT DEFAULTS 5-38
with WITHOUT NULL INPUT 5-38
Default form specification file
creating at system prompt 5-75
generating 1-14, 1-43, 5-73
modifying 1-14, 1-43, 5-73
DEFAULTS keyword
INPUT ARRAY statement 3-154
INPUT statement 3-131
DEFER statement 3-62
DEFINE section of REPORT statement 6-8
DEFINE statement
in a report 2-11
in FUNCTION statement 3-60, 3-113
in GLOBALS statement 3-60, 3-117
in MAIN statement 3-60
in REPORT statement 3-261
location 2-11
outside program blocks 3-60, 3-66
syntax and description 3-65
Delete key
deleting a line F-6, F-25
INPUT ARRAY statement 3-162
PICTURE attribute 5-49
DELETE keyword
INPUT ARRAY statement 3-160, 3-165
OPTIONS statement 3-230
DELETE statement, interrupting 3-236
Delimiter
changing in a screen form 5-68
for DATETIME values 3-302
for input file 3-184
for INTERVAL values 3-309
for output file 3-276
for screen fields 2-17
in a screen form 5-14
DELIMITER keyword
LOAD statement 3-184
UNLOAD statement 3-276
Demonstration application, listing A-30
Demonstration database
description of A-1
installing Intro-12
map of A-8
restoring 1-5, A-2
tables in A-3 to A-7
DIM attribute 3-290, 5-71, F-9
Dimensions of an array 3-73, 3-297
Disabled
form fields 5-6
menu options 2-16
DISPLAY ARRAY statement
ARR_CURR( ) 4-26
SET_COUNT( ) 4-80
syntax and description 3-85
DISPLAY ATTRIBUTE keywords 3-234
Display characteristics
background colors 4-56
default screen attributes 5-69
field attributes 5-28, 5-71
formatting output 3-77
output from a report 3-244, 6-14, 6-41
query by example 3-37
screen coordinates 5-12
table of color and intensity values 5-70
Display field
attributes 2-21, 3-37, 3-134, 3-156, 3-292, 5-27, 5-72
cursor movement 3-50
default attributes 5-21, 5-40, 5-55, 5-69
default field lengths 5-15, 5-75
delimiters 2-17, 5-14
display label 2-18
dividing character columns 5-25
field names 5-15, 5-20, 5-28
field tag 5-14, 5-33, 5-78
format 5-14
FORMONLY 5-20, 5-24
Help messages 2-21, 3-37, 3-134, 3-156
labels for 5-16
multiple-line fields 5-15
multiple-segment fields 5-57
names 5-21, 5-24
screen arrays 5-15
screen records 5-65
substring of a character column 5-22
THRU notation 3-363
verifying field widths 5-15
DISPLAY FORM statement 3-93
DISPLAY keyword
DISPLAY ARRAY statement 3-85
DISPLAY FORM statement 3-93
END DISPLAY statement 3-89
EXIT DISPLAY statement 3-89
OPTIONS statement 3-230
DISPLAY LIKE attribute 5-18, 5-28, 5-40
Display modes
Formatted mode 3-76, 3-80
Line mode 3-76
DISPLAY statement
CLIPPED 4-38
formatting 4-93
syntax and description 3-74
Distributed tables 5-24
Division (/) operator 3-321, 3-339, 3-357, 4-20, 4-22
Documentation notes Intro-10
Double data type (of C) C-35
DOWN keyword
SCROLL statement 3-268
syscolval table 5-70, B-7
DOWNSHIFT attribute 5-28, 5-41, 5-69, 5-70, B-7, E-10
DOWNSHIFT( ) 4-47
Drop option, PROGRAM Menu 1-22
Dynamic management statements 3-12

E

Editing keys
  CONSTRUCT statement 3-52
  INPUT ARRAY statement 3-173
  INPUT statement 3-147
  WORDWRAP fields 3-148, 5-60
Editor
  blanks in fields 5-59
  specifying with DBEDIT D-11
Ellipsis (. . .) symbol
  in code examples Intro-6
  in menu pages 2-15, 3-208
ELSE keyword, IF statement 3-124
END keyword
  ATTRIBUTES section of form 5-21
  END CASE statement 3-24
  END CONSTRUCT statement 3-47
  END DISPLAY statement 3-89
  END FOR statement 3-104
  END FOREACH statement 3-109
  END FUNCTION statement 3-116
  END GLOBALS statement 3-117
  END IF statement 3-124
  END INPUT statement 3-144, 3-169
  END MAIN statement 3-99
  END MENU statement 3-205
  END PROMPT statement 3-259
  END RECORD declaration 3-72, 3-313
  END REPORT statement 6-23
  END WHILE statement 3-288
  in a statement block 2-8
  INSTRUCTIONS section of form 5-63
  SCREEN section of form 5-13
  TABLES section of form 5-19
END statement 3-95
Endless loop 3-103, 3-285, 3-288
End-of-data condition 3-283
End-of-file character 3-346
ENTER key
  in ON KEY clause 3-41
  in query by example 3-52
ENVIGNORE environment variable D-30
Environment configuration file
  example D-2
  where stored D-3
Environment variable
  and case sensitivity D-4
  changing 4-83
  COLLCHAR E-2, E-4, E-5, E-11, E-16, E-18
  DBANSIWARN 2-25, D-8
  DBAPICODE E-2, E-11, E-23
  DBDATE 3-32, 3-182, 3-275, 3-300, 3-325, 4-45, 4-94, D-9, E-3, E-11, E-42, I-5
  DBDELIMITER 3-184, 3-276, D-11
  DBEDIT 1-10, 1-13, 1-24, 1-38, 1-42, 1-53, 5-74, D-11
  DBFLTMASK 3-32, 3-316
  DBFORM D-12, E-3, E-11, E-24
  DBFORMAT 3-32, 3-342, 4-91, D-14, E-3, E-5, E-11, E-35, E-38, E-39, E-41, E-42, I-5
  DBLANG B-5, D-18, E-3, E-11, E-24, E-42, I-5
  DBMONEY 3-32, 3-312, 3-325, 3-342, 4-91, D-21, E-3, E-5, E-11, E-35, E-38, E-41, E-42
  DBNLS E-5, E-11, E-16
  DBPATH 1-16, 1-26, 1-55, 3-33, 3-59, D-23
  DBPRINT 3-272, D-26
  DBREMOTECMD D-27
  DBSPACETEMP D-28
  DBTEMP 3-188, D-29
  DBTIME 3-32
  DBUPSPACE D-29
  default assumptions D-5
  defining in environment configuration file D-2
  definition of D-2
  ENVIGNORE D-30
  hierarchy of precedence E-12, E-31, E-33, E-34, E-37, E-38, E-40, E-41, E-42, E-48
  how to set in Bourne shell D-4
  how to set in C shell D-4
  how to set in Korn shell D-4
  INFORMIX environment variables, listing D-6
  INFORMIXCONRETRY D-30
  INFORMIXCONTIME D-31
  Informix-defined E-2, E-11, E-12
  INFORMIXDIR B-5, D-12, D-19, D-32, E-23, E-24
  INFORMIXSERVER D-33
  INFORMIXSHMBASE D-33
  INFORMIXSTACKSIZE D-34
  INFORMIXTERM D-35, F-1, F-20
  LANG D-20, E-2, E-11, E-12, E-14, E-15, E-21, E-23, E-25, E-31, E-34, E-40, E-42, E-43, E-50
  language and formatting E-11
  LC_COLLATE E-2, E-4, E-5, E-6, E-7, E-11, E-13, E-15, E-27
  LC_CTYPE E-2, E-5, E-6, E-7, E-11, E-15, E-23, E-29
  LC_MONETARY E-2, E-5, E-11, E-13, E-31, E-35, E-41, E-42
  LC_NUMERIC E-2, E-5, E-11, E-35, E-42
  listed D-6
  listed, for NLS D-7, D-39
  listed, for UNIX D-7
  meta-environment E-11
  NLS environment variables, listing D-7, D-39
  ONCONFIG D-36
  overriding a setting D-3, D-30
  PATH D-40
  PSORT_DBTEMP D-36
  rules of precedence D-6
  setting at the command line D-2
  setting in a shell file D-2
  SQLEXEC D-38
  SQLRM D-38
  SQLRMDIR D-39
  TERM D-41
  TERMCAP D-41
  TERMINFO D-42, F-20, F-26
  UNIX environment variables, listing D-7
  where to set D-2
  X/Open-defined E-11, E-12
Equal (=) sign
  Boolean expressions 3-329, 3-333, 4-31, 4-37

CONSTRUCT statement 3-49
FOR statement 3-102
LET statement 3-178
precedence of operators 3-328
Error
displaying 4-49
fatal 4-85
log file 2-27, 4-48, 4-51, 4-84
logging 4-51, 4-84
messages 1-33, 3-97, 4-48, 4-50, 4-84
messages, editing the 4glusr.msg file B-4
record 4-51, 4-84
Error handling
4GL built-in functions 4-48
compile-time errors 1-9, 1-38, 5-75
creating an error log 4-84
displaying error messages 4-49, 4-50
ERRORLOG( ) 4-51
fatal errors 2-26
logging error messages 4-51, 4-84
run-time errors 1-63, 4-84
SQLCA global record 2-23
STARTLOG( ) 4-84
untrappable errors 2-26
with status variable 4-48, 4-49, 4-50
ERROR keyword
ERROR statement 3-96
OPTIONS statement 3-229
WHENEVER statement 3-177, 3-281
Error line 2-19, 3-51, 3-94, 3-96, 3-172, 3-227, 4-49, 4-50
ERROR LINE keywords in OPTIONS statement 3-229
ERROR statement 3-96
ERRORLOG( ) 4-51
ERR_GET( ) 4-48
ERR_PRINT( ) 4-49
ERR_QUIT( ) 4-50
Escape character
in input files 3-184
in output files 3-276
ESCAPE keyword
with LIKE 3-336, 4-34
with MATCHES 3-336, 4-34
ESQL/C functions 1-30, 1-62
EVERY ROW keywords
default format of a report 3-261, 6-23
ON EVERY ROW control block 6-34
Exceptional conditions
end of data 3-283
in evaluating expressions 3-283
SQL errors 3-282, 4-85
warnings 3-61, 3-281
Exceptions
handling with DEFER 2-23
handling with WHENEVER 2-23
WHENEVER statement 3-281
Exclamation (!) point
Boolean expressions 3-329, 3-333, 4-31, 4-37
precedence of operators 3-328
PROGRAM attribute 5-51
EXCLUSIVE keyword of DATABASE 3-61
Exclusive mode, DATABASE statement 3-61
Executable statements 3-66, 3-119
EXECUTE PROCEDURE keywords in INSERT statement 3-184
EXECUTE statement in query by example 3-32
EXISTS keyword 3-330
Exit code 3-99, 3-266
EXIT keyword
EXIT CASE statement 3-24
EXIT CONSTRUCT statement 3-47
EXIT DISPLAY statement 3-89
EXIT FOR statement 3-104
EXIT FOREACH statement 3-109
EXIT INPUT statement 3-143, 3-151, 3-169, 3-175
EXIT MENU statement 3-202, 3-210
EXIT PROGRAM statement 3-98, 3-192
EXIT WHILE statement 3-288
in a statement block 2-8
versus GOTO statement 3-122
Exit option
FORM Menu 1-16, 1-45
MODULE Menu 1-12, 1-40
PROGRAM Menu 1-22, 1-51
EXIT statement 3-98
Explicit NLS environment
defined E-7, E-19
disadvantages of E-7
example E-20
mentioned E-4, E-5, E-18
Exponent
DECIMAL data type C-23, C-25
FLOAT data type 3-306, 3-341, 4-21
SMALLFLOAT data type 3-316, 3-341, 4-21
Exponentiation (* *) operator 3-339, 3-341, 4-20, 4-21
Expressions
in form specifications 5-32
in SQL statements 3-330, 4-9
in syscolatt table 5-71
Expressions in 4GL statements
arithmetic expressions 3-339, 4-18
Boolean expressions 3-333, 4-30
character expressions 3-343
data type conversion 3-319, 3-324, 3-340, 3-357, 4-22, 4-54
expression types 3-326
field operators 4-37, 4-64, 4-66, 4-69
integer expressions 3-338
number expressions 3-341
operands 3-327, 3-331
operators 3-327, 4-10, 4-11, 4-12
parentheses 3-327
resetting status 3-283
time expressions 3-347, 4-54
EXTEND( ) operator
implicit 3-325
in arithmetic expressions 3-352, 4-54
syntax and description 4-53
Extension checking, specifying with DBANSIWARN D-8
External
editor 5-50, 5-76
table, query by example 3-35

EXTERNAL keyword, REPORT statement 3-262, 6-22

F

FALSE (Boolean constant) 3-62, 3-333, 5-31
Fatal errors 2-26
FETCH statement
  implicit with FOREACH 3-105
  interrupting 3-236
  NOTFOUND code 2-24
  with Update cursors 3-108
  with WHENEVER 3-283
fgiusr.c file 1-65
fglapi.h C-12
fgldb command 1-64
fglgo command 1-5, 1-48, 1-62, 3-267, 4-16
fglpc command 1-5, 1-59
fgl_call( ) macro C-19
FGL_DRAWBOX( ) 4-56
fgl_end( ) macro C-22
fgl_exitfm( ) macro C-21
FGL_GETENV( ) 4-58
FGL_KEYVAL( ) 4-60
FGL_LASTKEY( )
  syntax and description 4-62
  with CONSTRUCT 3-47
  with DISPLAY ARRAY 3-91
  with INPUT 3-144
  with INPUT ARRAY 3-170
fgl_start( ) macro C-17
Field
  buffer 3-47, 3-91, 3-144, 3-170, 4-66
  clause 3-359, 4-64
  data type 5-21, 5-25
  description 3-292, 5-20, 5-72
  disabling 5-6
  editing keys 3-52, 3-147, 3-173
  labels 5-15, 5-76
  length 5-14
  multiple-segment 5-26, 5-57
  names in screen forms 4-69, 5-19, 5-21, 5-24
  operators 3-328, 4-37, 4-64, 4-66, 4-69
  qualifier 5-22
Field attributes
  description 5-27
  interacting with users 2-18
  order of precedence 5-72
  query by example 3-37
FIELD keyword
  AFTER FIELD 3-42
  BEFORE FIELD 3-40, 3-137, 3-161
  CONSTRUCT statement 3-40, 3-42, 3-44
  INPUT ARRAY statement 3-164
  INPUT statement 3-139
  NEXT FIELD 3-44
  OPTIONS statement 3-230
FIELD ORDER CONSTRAINED keywords 3-51, 3-232
FIELD ORDER UNCONSTRAINED keywords 3-51, 3-232
Field tag
  in Boolean expressions 3-331, 5-33, 5-78
  in default forms 5-15, 5-76
  in the ATTRIBUTES section 5-20, 5-33
  in the SCREEN section 5-14, 5-57
  naming conventions 2-10, 5-15
FIELD_TOUCHED( ) operator
  syntax and description 4-64
  using in SQL expressions 3-330
  with CONSTRUCT 3-45, 3-47
  with DISPLAY ARRAY 3-91
  with INPUT 3-144
  with INPUT ARRAY 3-170
File
  environment configuration D-2
  shell D-2
  temporary for OnLine D-28
  temporary for SE D-29
File extensions
  .4be 1-33, 1-71
  .4bl 1-33, 1-71
  .4bo 1-33, 1-71
  .4ge 1-9, 1-11, 1-18, 1-32, A-30
  .4gi 1-47, 1-53, 1-56, 1-61, 1-62, 1-71
  .4gl 1-10, 1-24, 1-30, 1-32, 1-52, 1-59, 1-71, A-30
  .4go 1-48, 1-53, 1-58, 1-59, 1-62, 1-71
  .c 1-19, 1-33, 1-67
  .dbs 3-58, 3-59, 5-10
  .ec 1-19, 1-26, 1-30, 1-33, 1-67
  .erc 1-33, 1-71
  .err 1-28, 1-33, 1-58, 1-71, 5-75
  .fbm 1-33, 1-71
  .frm 1-33, 1-71, 5-74, 5-75, D-12
  .h 1-69, C-23
  .iem D-18
  .msg B-4
  .o 1-8, 1-24, 1-32
  .out 1-31, 1-68
  .pbr 1-33, 1-71
  .per 1-13, 1-33, 1-71, 5-75
  .src A-30
  ˙4gl 2-6, 3-66, 3-119
  ˙frm 3-217, 3-221
  ˙per 3-217
FILE keyword
  LOCATE statement 3-188
  OPTIONS statement 3-230, 3-234, B-2
  PRINT statement 3-254, 6-44
Filename
  LOAD statement 3-181
  UNLOAD statement 3-274
Fill character
  ampersand 4-92
  dollar sign 4-92
  parentheses 4-92
  pound sign 4-92, 5-43
FINISH REPORT statement 3-100, 6-5
FIRST keyword
  OPEN WINDOW statement 3-223, 3-226
  OPTIONS statement 3-231
  REPORT statement 6-33
FIRST PAGE HEADER control block 3-271, 6-33, 6-37
Fixed-point numbers 3-306, 3-316, 3-342
FLOAT data type
  data type conversion 3-319, 3-324
  declaration 3-68, 3-294
  description 3-305
  display fields 5-43

display width 3-78, 5-76, 6-44
in NLS E-9
literal values 3-306, 3-342
Float data type (of C) C-33
Floating-point numbers 2-25, 3-306, 3-316, 3-320, 3-342, C-32
FOR keyword
CONTINUE FOR statement 3-55, 3-103
DECLARE statement 3-105
END FOR statement 3-104
EXIT FOR statement 3-104
FOR statement 3-102
PROMPT statement 3-257
FOR statement 3-102
FOREACH keyword in CONTINUE FOREACH statement 3-55
FOREACH statement
interrupting 3-236
syntax and description 3-105
Foreground colors 4-56, F-18
Form
binding fields to variables 5-4
binding to the database 5-4
clearing 3-26
closing 3-29, 3-30
declaring 3-218
dimensions 5-12
displaying 3-217, 3-228
fields 3-359, 5-4
identifying the current field 4-69
line 2-19, 3-94, 3-226
naming conventions 2-10
screen records 5-64
syntax of form specification 5-8
FORM Design Menu 1-12
FORM keyword
CLEAR FORM statement 3-26
CLOSE FORM statement 3-29
DISPLAY FORM statement 3-93
OPEN FORM statement 3-217
OPEN WINDOW statement 3-29, 3-223
OPTIONS statement 3-229, 3-234
FORM LINE keywords
OPEN WINDOW statement 3-223
OPTIONS statement 3-229
Form management blocks
CONSTRUCT statement 3-38
INPUT ARRAY statement 3-157
INPUT statement 3-135
Form specification file
ATTRIBUTES 5-7, 5-20
DATABASE 5-6, 5-10
DISPLAY FORM statement 3-292, 5-72
INSTRUCTIONS 5-7, 5-63
multiple tables 5-18
OPEN FORM statement 3-217
OPEN WINDOW statement 3-221
overview 5-3
PERFORM forms 5-77
SCREEN 5-6, 5-12
TABLES 5-6, 5-18
Form specification file, using
correcting errors 1-14
creating 5-75
default form specification file 5-75
generating 1-12, 1-41
graphics characters 5-16
multiple tables in 5-73
FORM4GL
attribute syntax 5-28
command line syntax 5-75
creating a default form specification file 5-73
default attributes 5-21
default field tags 5-75
default screen records 5-64
description 5-3
field attributes 5-27
file extensions created by 5-75
from Programmers Environment 1-15, 1-44
graphics characters in screen section 5-16
verifying field widths 5-15
Format
date data D-9, D-18, E-47
monetary data D-14, E-31, E-45
numeric data D-14, E-35, E-45
FORMAT attribute
in fields 5-28
in NLS E-10, E-31, E-33, E-37, E-40, E-42, E-44
syntax and description 5-42
FORMAT keyword
FORMAT attribute 5-43
REPORT statement 3-261
FORMAT section of REPORT statement
AFTER GROUP OF 6-29
BEFORE GROUP OF 6-31
CLIPPED 6-44
COLUMN 6-37
COLUMN operator 4-40
EVERY ROW 6-24
FIRST PAGE HEADER 6-33
NEED statement 3-216, 6-40
ON EVERY ROW 6-34
ON LAST ROW 6-36
PAGE HEADER 6-37
PAGE TRAILER 6-38
PAUSE statement 3-244, 6-41
PRINT statement 3-254, 6-42
SKIP statement 3-269, 6-52
syntax 6-23
USING 6-44
WORDWRAP 4-102, 6-50
Format strings
in syscolatt table 5-71
with FORMAT attribute 5-42, 5-48, B-9
with PICTURE attribute 5-48
with USING operator 3-343, 4-91
Formatted mode 3-76, 3-80
Formatting
data 2-18, 5-27
number expressions 4-91
Formatting a report
automatic page numbering 6-37
default report format 6-15, 6-24
formatting dates 4-94
formatting numbers 4-91
grouping data 4-14, 6-46
page headers and trailers 6-33, 6-37, 6-38
printing column headings 6-37
setting margins 4-102, 6-11, 6-12, 6-14, 6-16, 6-50
setting page eject character 6-17
setting page length 6-12

skipping to top of page 3-269
starting a new line 3-269, 4-103, 6-44, 6-51
starting a new page 3-216, 3-269, 4-103, 6-17, 6-40, 6-51, 6-52
FORMFEED character in TEXT values 3-317, 3-345
FORMONLY field 3-48, 3-130, 3-299, 5-24, 5-29
FORMONLY keyword
ATTRIBUTES section 5-24
CLEAR statement 3-28
CONSTRUCT statement 3-48
DATABASE section 5-18
field clause 3-359
INSTRUCTIONS section 5-65
THRU keyword 3-363
FOUND keyword in WHENEVER statement 3-281
FRACTION keyword
CURRENT operator 4-42
DATETIME qualifier 3-301, 3-349, 4-42
INTERVAL qualifier 3-308, 3-353
UNITS operator 4-89
FREE statement 3-190
FROM keyword
CONSTRUCT statement 3-35
INPUT ARRAY statement 3-153
INPUT statement 3-132
LOAD statement 3-182
OPEN FORM statement 3-217
SELECT statement 3-275
Function keys 1-47, F-5, F-24
FUNCTION statement 3-111
Functions
as arguments 3-113, 3-332
built-in 4GL functions 4-5, 4-12
built-in SQL functions 4-6
C language 1-30, 1-62, 4-6, C-23
dummy functions 3-115
ESQL/C language 4-7
function calls in expressions 3-332
function calls in reports 6-45
INFORMIX-ESQL/C 1-30, 1-62
invoking with CALL 3-16
invoking with WHENEVER 3-281
naming conventions 2-10
overview 4-5
programmer-defined 3-111
prototypes 3-112, 4-7

G

Generate option, FORM Menu 1-14, 1-43
GET_FLDBUF( ) operator 3-45, 3-330, 4-66
Global
aggregate functions 3-262
Language Support E-4
Source array 1-48
Global variables
declared in MAIN 3-192
declaring 3-66, 3-117
importing 3-119
scope of reference 2-11
GLOBALS keyword
END GLOBALS statement 3-119
GLOBALS statement 3-117
GLOBALS statement
syntax and description 3-117
with DATABASE 3-60, 3-119
with DEFINE 3-66, 3-117
GOTO keyword, WHENEVER statement 3-177, 3-281
GOTO statement 3-122
GRANT statement
with LOAD 3-181
with UNLOAD 3-274
Graphics characters in forms 5-16
Greater than (>) symbol
BYTE values in reports 6-44
COLOR attribute 5-32
relational operator 3-48, 3-329, 3-334, 4-32, 4-37
REVERSE attribute 5-53
GREEN attribute 3-290, 5-31, 5-71, F-20
GROUP keyword
AFTER GROUP OF control block 6-29
aggregate functions 3-262, 4-14, 6-31, 6-46
BEFORE GROUP OF control block 6-31

H

Header files, decimal.h C-23
HEADER keyword
FIRST PAGE HEADER control block 6-33
PAGE HEADER control block 6-37
Help
menu 4-81, B-4
window 2-22, 4-81
Help file
compiling with mkmessage B-2
showhelp function B-4
HELP keyword
CONSTRUCT statement 3-37
INPUT ARRAY statement 3-155
INPUT statement 3-133
MENU statement 3-197
OPTIONS statement 3-230, 3-234, B-2
PROMPT statement 3-258
Help message
creating and compiling 1-8, 1-36, B-2
displaying 1-7, 1-35, 3-37, 3-133, 3-155, 4-81
specifying Help file 3-230
using SHOWHELP( ) 4-81
Hexadecimal numbers 3-184, 3-276
Hidden menu options 3-203
HIDE keyword, MENU statement 3-203
HOLD keyword in DECLARE statement 3-105
Host system 3-362
HOUR keyword
DATETIME qualifier 3-301, 3-349, 4-42
INTERVAL qualifier 3-309, 3-353
UNITS operator 4-89
Hyphen (-) symbol
comment indicator 2-6
DATETIME separator 3-302, 3-351

in window border 5-16, F-6
INTERVAL separator 3-309, 3-355
with CONSTRUCT 3-50
with MATCHES 3-335, 4-34

I

i4gl command 1-5, 1-23, 5-73
i4gldemo script 1-5
Icons in syntax diagrams Intro-7
Identifier
database cursor 3-106
declaring 2-9
function arguments 3-113
function name 3-112
naming conventions 2-9
predefined 2-12
report arguments 3-261
report name 3-100, 3-271
scope of reference 2-11
SQL identifiers 2-10, 2-12
Identifiers in NLS E-5, E-8, E-21, E-29
IF statement 3-124
Implicit
mapping E-4, E-6, E-8, E-18
names, declaring 3-72
Implicit NLS environment
advantages of E-18
defined E-6, E-19
example E-20
mentioned E-4, E-5
IN keyword
Boolean expressions 3-337, 4-35, 5-32, 5-71
CREATE TABLE statement 3-70
LOCATE statement 3-70, 3-187
INCLUDE attribute 5-28, 5-44, 5-70
INCLUDE keyword in syscolval table 3-279
Incompatible data types 3-324
Indirect typing 3-59, 3-69, 3-119, 6-8
INFIELD( ) operator
field-level Help 3-37, 3-134, 3-156
Help messages 2-21
in ON KEY clause 3-139, 3-164
in SQL expressions 3-330
syntax and description 4-69
with CONSTRUCT 3-47
with DISPLAY ARRAY 3-91
with INPUT 3-144
with INPUT ARRAY 3-170
infocmp utility F-25, F-29
.informix environment configuration file D-2
Informix extension checking, specifying with DBANSIWARN D-8
informix owner name 4-9
INFORMIX-4GL
as a report writer 6-3
command file names 1-58
language overview 2-3
screen forms 5-3
versions 1-3
INFORMIXCONRETRY environment variable D-30
INFORMIXCONTIME environment variable D-31
Informix-defined environment variable E-2, E-11, E-12
INFORMIXDIR environment variable B-5, D-12, D-19, D-32, E-23, E-24
INFORMIX-ESQL/C 2-7
INFORMIX-ESQL/C functions 1-30, 1-62, 2-7, 4-7
INFORMIX-OnLine engine
database names 2-10
interrupting SQL statements 3-236
rolling back transactions 3-238
specific data types 3-14
INFORMIX-OnLine/Optical statements 3-13
INFORMIX-SE engine
database names 2-10
interrupting SQL statements 3-236
rolling back transactions 3-238
specific data types 3-14
INFORMIXSERVER environment variable D-33
INFORMIXSHMBASE environment variable D-33
INFORMIX-SQL
Interactive Editor 5-62
screen forms 5-78
INFORMIXSTACKSIZE environment variable D-34
INFORMIXTERM environment variable D-35, F-1, F-20
informix.rc file D-2
INITIALIZE statement 3-125
INPUT ARRAY statement
ARR_CURR( ) 4-26
SCR_LINE( ) 4-78
SET_COUNT( ) 4-80
syntax and description 3-152
INPUT ATTRIBUTE keywords 3-234
Input file
dbload utility 3-183
LOAD statement 3-182
INPUT keyword
AFTER INPUT block 3-140, 3-166
BEFORE INPUT block 3-136, 3-159
CONTINUE INPUT 3-143, 3-169
CONTINUE INPUT statement 3-55
EXIT INPUT statement 3-143, 3-169
INPUT ARRAY statement 3-152
INPUT statement 3-128
OPTIONS statement 3-230
WITHOUT NULL INPUT 5-10
INPUT NO WRAP keywords 3-232
Input record 3-182, 3-242, 4-80, 6-5
INPUT statement
ARR_COUNT( ) 4-24
syntax and description 3-128
INPUT WRAP keywords 3-232
Insert
editing mode 3-52, 3-147, 3-173
privilege 3-181
Insert key
defining F-6, F-25
INPUT ARRAY statement 3-162
INSERT keyword
GRANT statement 3-181
INPUT ARRAY statement 3-160, 3-165
LOAD statement 3-184

OPTIONS statement 3-230
INSERT statement
DATETIME or INTERVAL values 3-322
interrupting 3-236
NOENTRY attribute 5-47
with INPUT 3-128
with INPUT ARRAY 3-152
with LOAD 3-184
Installation directory, specifying with INFORMIXDIR D-32
INSTRUCTIONS section of form specification
screen arrays 5-66
SCREEN RECORD keywords 3-359, 4-64, 5-63, 5-64, 5-66
screen records 5-63
syntax 5-63
INT data type 3-306
Int data type (of C) C-28, C-29
Integer
division 3-339, 4-22
expression 3-338
literal 3-340
INTEGER data type
data type conversion 3-319, 3-324
declaration 3-68, 3-294
description 3-306
display fields 5-12
display width 3-78, 5-76, 6-44
in report output 6-44
literal values 3-342
Intensity attributes 5-27, 5-70, D-35
Intentional blanks in multiple-segment fields 3-147, 5-59
Interactive Debugger
Debugger path 1-48
description of 1-63
invoking 1-36, 1-51
International application development
preparing a translation checklist I-6
requirements for I-3
Internationalization I-2
Interrupt key
interrupting SQL statements 3-64, 3-235, 3-237
with CONSTRUCT 3-41
with DEFER 3-63
with DISPLAY ARRAY 3-92
with INPUT 3-151
with INPUT ARRAY 3-175
with MENU 3-210
with PROMPT 3-258
INTERRUPT keyword
CONSTRUCT statement 3-41
DEFER statement 3-62
MENU statement 3-199
OPTIONS statement 3-64, 3-231, 3-235, 3-237
Interrupt signal 2-23, C-15
INTERVAL data type
arithmetic operations 3-357, 4-22, 4-54
as character string 3-182, 3-311, 3-322
data type conversion 3-324
declaration 3-68, 3-294, 3-307
description 3-307
display fields 5-39, 5-49
display width 3-78, 5-76, 6-44
in report output 6-44
in time expressions 3-357, 4-22
literal 3-275, 3-309, 3-355
qualifiers 3-307, 3-353, 5-39
time data types 3-295
values 3-348
INTO keyword
FOREACH statement 3-107
INSERT statement 3-301, 5-4
LOAD statement 3-184
SELECT statement 3-32, 3-107, 3-187, 3-317
int_flag 3-41, 3-52, 3-62, 3-199, 3-236, 3-258
Inverse video 3-224, 5-53, 5-70
INVISIBLE attribute 3-83, 3-214, 3-290, 3-291, 5-46, 5-71
Invisible menu options 3-200, 3-205
Invoking
4GL Compiler 1-5, 1-29, 1-30, 1-59
4GL programs 1-5, 1-27, 1-50
FORM4GL 1-15, 1-44, 5-75
Interactive Debugger 1-36, 1-56
Programmers Environment 1-5, 1-6, 1-34
IS keyword
CURRENT WINDOW statement 3-56
IS NULL operator 4-37
NULL test 3-336, 4-33, 5-33
ISO 8859 character sets I-2
items table in stores database A-4

J

Join columns 3-73, 5-78
Joins in the stores database columns A-8
Jump instructions 2-7, 3-55, 3-122, 3-177, 3-284
Justified data display
left justified 3-80, 3-86, 4-92, 4-103, 5-31, 6-51, C-26
right justified 3-80, 3-86, 4-93, 5-42

K

Key
activation key 3-198
assigning logical functions 3-235
choosing menu options 3-206
Help 2-21
Help key B-2
Interrupt 3-64, 3-235
scrolling and editing 3-86, 3-88, 3-147, 3-173
KEY keyword
ACCEPT KEY 3-230
CONSTRUCT statement 3-41
DELETE KEY 3-173, 3-230
DISPLAY ARRAY statement 3-87
HELP KEY 3-230
INPUT ARRAY statement 3-161
INPUT statement 3-137
INSERT KEY 3-173, 3-230
MENU statement 3-199
NEXT KEY 3-230
PREVIOUS KEY 3-230
PROMPT statement 3-258
Keystroke buffer 3-47, 3-91, 3-144, 3-170

Keywords
as identifiers 2-10
typographic convention Intro-5
Korn shell
how to set environment variables D-4
.profile file D-2

L

LABEL statement
syntax and description 3-177
with GOTO 3-122
with WHENEVER 3-284
LANG environment variable
effect of where set on precedence E-14
interaction with DBLANG D-20
lack of standardization E-26
mentioned E-2, E-11, E-12, E-15, E-21, E-23, E-31, E-34, E-40, E-42, E-43, E-50
setting default for LC_ variables E-2, E-13, E-25
syntax E-25
Language
and formatting environment variable E-11
supplement E-24, E-26, E-50
Language features
built-in functions 4-6
built-in operators 4-10
flat-file input 3-181
functions 4-5
statement types 3-11, 3-13
Large binary data types 3-70, 3-149, 3-174, 3-186, 3-296
LAST keyword
OPEN WINDOW statement 3-223, 3-226
OPTIONS statement 3-231
REPORT statement 3-261, 6-36
LC_ variable
defined E-11
effect of where set on precedence E-14
lack of standardization E-2, E-12, E-15, E-25, E-40
LC_COLLATE environment variable
database storage of value E-6, E-7, E-15
defined E-27
mentioned E-2, E-4, E-5, E-6, E-11, E-13
syntax E-27
LC_CTYPE environment variable
database storage of value E-6, E-7, E-15, E-29
defined E-29
interaction with DBAPICODE E-23
mentioned E-2, E-5, E-11
syntax E-29
LC_MONETARY environment variable
defined E-31
mentioned E-2, E-5, E-11, E-13, E-35, E-41, E-42
syntax E-5, E-32
LC_NUMERIC environment variable
defined E-35
mentioned E-2, E-5, E-11, E-42
syntax E-36
Leading currency symbol D-14, D-21, E-33, E-40, E-45
LEFT attribute 5-31
LEFT MARGIN keywords 6-12
Left margin of a 4GL window 3-219
LENGTH keyword, PAGE LENGTH clause 6-12
LENGTH( ) 4-71
Less than (<) symbol
BYTE values in reports 6-44
COLOR attribute 5-32
in USING format strings 4-92
relational operator 3-48, 3-334, 4-32, 4-37
REVERSE attribute 5-53
LET statement
CLIPPED operator 4-38
conversion of MONEY to CHAR E-33, E-37
in NLS E-10, E-47
syntax and description 3-178
USING operator 4-91
Letter case sensitivity 2-4, 2-10, 4-6, 4-94, 5-15
Library functions
decadd( ) C-36
deccmp( ) C-37
deccopy( ) C-38
deccvasc( ) C-25
deccvdbl( ) C-34
deccvflt( ) C-32
deccvint( ) C-28
deccvlong( ) C-30
decdiv( ) C-36
dececvt( ) C-39
decfcvt( ) C-39
decmul( ) C-36
decsub( ) C-36
dectoasc( ) C-26
dectodbl( ) C-35
dectoflt( ) C-33
dectoint( ) C-29
dectolong( ) C-31
LIKE keyword
Boolean expressions 5-71
DEFINE statement 3-60, 3-69
DISPLAY LIKE attribute 5-18, 5-40
FORMONLY fields 5-18, 5-24
INITIALIZE statement 3-126
RECORD data type 3-72, 3-313, 3-314
string operator 3-328, 3-335, 4-33
VALIDATE LIKE attribute 5-18, 5-55
VALIDATE statement 3-279
LINE keyword
OPEN WINDOW statement 3-223
OPTIONS statement 3-229
SKIP statement 3-269, 6-52
Line mode 3-76, 3-77
Line mode overlay 4-40
Line number
in a program array 4-26
in a screen array 4-78, 5-63, 5-66
LINEFEED characters between statements Intro-8, 2-3
Linefeed key in ON KEY clause 3-41
LINENO operator 3-254, 4-73, 6-48

LINES keyword
NEED statement 3-216, 6-40
SKIP statement 3-269, 6-52
Link-time errors 3-16
Literal values
DATE data type 3-275, 3-349
DATETIME data type 3-275, 3-351
integers 3-340
INTERVAL data type 3-275, 3-355
numbers 3-275, 3-342
LOAD statement
in NLS E-7, E-10, E-15, E-35, E-40, E-42
interrupting 3-236
specifying field delimiter with DBDELIMITER D-11
syntax and description 3-181
Local variables 2-11, 2-13, 3-66, 3-114, 3-192, 3-261, 6-8
Locale
database locale E-2
defined E-2
file E-26
mentioned E-15, E-25
user locale E-2
variable E-11, E-12
Locale-sorted columns E-6, E-8
Localization
of applications I-10
process of I-2
LOCATE statement 3-186, 3-317
LOCK TABLE statement with LOAD 3-185
Logging
error messages 4-84
transactions 3-104, 3-108, 3-287, 4-83
Logical operators 3-329, 3-333, 4-31, 4-37
Long data type (of C) C-30, C-31
LOOKUP attribute of PERFORM 5-78
Loops
FOR statement 3-102
FOREACH statement 3-105
in syntax diagrams Intro-8
using CONTINUE 3-55
WHILE statement 3-287
Lowercase characters
DOWNSHIFT attribute 5-41, B-7
DOWNSHIFT( ) 4-47
in field tags 5-15
in identifiers 2-4, 2-10
names of C functions 4-6
SHIFT attribute 5-70, B-7
UPSHIFT attribute 5-54, B-7
UPSHIFT( ) 4-90

M

m symbol in format strings 4-94, 5-42
MAGENTA attribute 3-290, 5-31, 5-71, F-20
MAIN statement
in source-code modules 1-31
preceded by DATABASE 3-61
syntax 3-191
uniqueness 2-7
Mantissa
DECIMAL data type C-23
FLOAT data type 3-306
SMALLFLOAT data type 3-316
manufact table in stores database A-6
Mapping file E-23, E-24
MARGIN keyword
BOTTOM MARGIN clause 6-11
LEFT MARGIN clause 6-12
RIGHT MARGIN clause 6-14, 6-50
TOP MARGIN clause 6-16
WORDWRAP operator 4-102, 6-50
MATCHES keyword
Boolean operator 4-33
description 3-335
in syscolatt table 5-71
precedence 3-328
with COLOR attribute 5-31
Maximum size of VARCHAR data type 3-68, 3-294
MAX( ) aggregate function 2-25, 6-47
MDY( ) operator 4-74
Member
of input record 3-242
of program record 3-72, 3-313
MEMORY keyword in LOCATE statement 3-187
Memory management
CLOSE FORM statement 3-29
CLOSE WINDOW statement 3-30
FREE statement 3-190
Large variables 3-190
Menu
form file E-24
help line 2-15, 2-17, 3-199
line 2-17, 3-94, 3-193, 3-226
options of Programmer's Environment 1-6, 1-27
MENU keyword
BEFORE MENU clause 3-196
CONTINUE MENU statement 3-55, 3-201
END MENU statement 3-205
EXIT MENU statement 3-202
OPEN WINDOW statement 3-223
OPTIONS statement 3-229
MENU LINE keywords
OPEN WINDOW statement 3-223
OPTIONS statement 3-229
Menu options
disabled 2-17, 3-203, 3-209
hidden 2-16, 3-203
invisible 2-16, 3-200
MENU statement 3-193
Menus of 4GL
in a national language D-12
MENU statement 2-15
menu title 2-15
nested 2-16
Message
file D-18, E-24, E-47
line 2-19, 3-94, 3-213, 3-226
numbers in help files B-2
MESSAGE keyword
MESSAGE statement 3-213
OPEN WINDOW statement 3-223
OPTIONS statement 3-230

MESSAGE LINE keywords
  OPEN WINDOW
    statement 3-223
  OPTIONS statement 3-230
MESSAGE statement 3-213
Meta-environment variable E-11
Method 4-5
Minus (-) sign
  comment indicator 2-6
  DATETIME separator 3-303
  in format strings 4-92
  in INTERVAL literals 5-39
  in literal numbers 3-342, C-25
  in window border 5-16, F-6
  INTERVAL literals 3-309, 3-355
  INTERVAL separator 3-311
  OPEN WINDOW
    statement 3-226
  OPTIONS statement 3-231
  subtraction operator 3-226, 3-231,
    3-328, 3-339, 3-357, 4-20, 4-22
  unary operator 3-328, 3-340
MINUTE keyword
  DATETIME qualifier 3-301,
    3-349, 4-42
  INTERVAL qualifier 3-309, 3-353
  UNITS operator 4-89
MIN( ) aggregate function 2-25,
  4-14, 6-47
mkmessage utility 1-8, 1-36, 3-234,
  B-2
Modify option
  FORM Menu 1-13, 1-42, 5-73
  MODULE Menu 1-8, 1-36
  PROGRAM Menu 1-18, 1-46
Modular scope operator 2-14
Module
  compiling 1-11, 1-39
  option of INFORMIX-4GL
    Menu 1-7, 1-35
  running multi-module
    programs 1-11, 1-40, 1-63
  variables 2-11, 3-66, 3-192
MODULE Menu 1-23, 1-52
Modulus (MOD) operator 3-328,
  3-339, 3-341, 3-356, 4-20, 4-21
MONEY data type
  data type conversion 3-319, 3-324
  declaration 3-68, 3-294
  default value 5-38
  description 3-312
  display width 3-78, 5-76, 6-44
  formatting with
    DBFORMAT 4-91, E-47
  in input files 3-182
  in NLS E-9
  in output files 3-275
  in report output 6-44
  literal values 3-342
Monochrome terminals 3-291
Monospace typeface Intro-5
MONTH keyword
  CURRENT operator 4-42
  DATETIME qualifier 3-301,
    3-349, 4-42
  EXTEND( ) operator 4-54
  INTERVAL qualifier 3-309, 3-353
  MONTH( ) operator 4-75
  UNITS operator 4-89
MONTH( ) operator 4-75
Multiple-form programs 3-56
Multiple-module programs,
  compiling 1-11, 1-24, 1-39, 1-47,
  1-53, 1-62
Multiple-segment fields
  description of 5-26
  in WORDWRAP fields 5-57
  with CONSTRUCT 3-51
  with INPUT 3-148
Multiple-statement PREPARE 3-61
Multiple-table
  forms 5-18
  screen records 5-67
  views 5-29
Multiplication (*) operator 3-300,
  3-321, 3-339, 3-356, 4-20, 4-21

N

NAME keyword
  CONSTRUCT statement 3-34
  DISPLAY statement 3-74
  INPUT statement 3-131
Name scope 2-11
Named values 3-331
Naming conventions
  display fields 5-21, 5-24, 5-29
  field tags 5-15
Naming rules
  4GL identifiers 2-10
  databases 3-58
  SQL identifiers 2-9
Native Language Support
  classification of variables E-11
  database access restrictions E-7
  defined E-2
  environment variables listed D-7,
    D-39
  features supported in Version
    6.0 E-8
  multiple locales E-49
NCHAR data type E-4, E-6, E-9,
  E-18
NEED statement 3-216, 6-40
Network environment variable
  SQLRM D-38
  SQLRMDIR D-39
New option
  FORM Menu 1-15, 1-44
  MODULE Menu 1-10, 1-38
  PROGRAM Menu 1-20, 1-49
NEWLINE character
  in TEXT values 3-182, 3-275,
    3-317, 3-345
  in VARCHAR values 3-182, 3-275
  in WORDWRAP fields 3-148,
    5-59
  input record separator 3-182
  output record separator 3-274
  report output 6-18, 6-51
Next
  key F-6, F-25
  menu option 4-81
NEXT FIELD keywords
  CONSTRUCT statement 3-44
  INPUT ARRAY statement 3-167
  INPUT statement 3-141
NEXT keyword
  CONSTRUCT statement 3-44
  INPUT ARRAY statement 3-168
  INPUT statement 3-142
  MENU statement 3-203
  OPTIONS statement 3-230
Next Page key
  DISPLAY ARRAY statement 3-91
  INPUT ARRAY statement 3-162

NEXTPAGE keyword 3-234
NLS database
  defined E-5
  mentioned E-16
  performance penalty E-18
NLS environments
  defined E-5
  distinctions between E-6
  summary of E-8
NO keyword
  OPTIONS statement 3-230
  syscolval table 5-70
NOENTRY attribute 5-47
Nondestructive backspace 3-172
Non-local database 3-362
Non-NLS database
  defined E-6
  mentioned E-16
  US English NLS sorting different from E-22
  when preferable to NLS database E-7
Non-NLS environment, defined E-4, E-19
Non-significant characters 2-3
NORMAL attribute 3-290, 5-71
Normalized form of a DECIMAL number C-23
NOT FOUND keywords in WHENEVER statement 3-281
NOT keyword
  Boolean operator 3-329, 3-333, 4-31, 4-37
  NULL test 3-332, 3-336, 4-33
  precedence of operators 3-328
  range test 4-35
  set membership test 3-337, 4-35
  WHENEVER statement 3-283
NOT NULL keywords
  COLOR attribute 5-33
  FORMONLY fields 5-24, 5-25
NOTFOUND keyword
  contrasted with NOT FOUND keywords 3-284
  status after SELECT 2-24
  with FOREACH 3-105
NOUPDATE attribute of PERFORM 5-78
NULL keyword
  COLOR attribute 5-32
  DATABASE section 5-10
  FORMONLY fields 5-25
  INCLUDE attribute 5-44
  INITIALIZE statement 3-127, 3-364
  IS NULL operator 3-337
  LET statement 3-178, 3-179
NULL values
  aggregate functions 2-25, 4-14, 6-46
  as default 3-130, 3-154
  in ASCII files 3-182, 3-275
  in Boolean expressions 3-103, 3-336, 4-33, 5-33
  in display fields 5-24, 5-25, 5-38, 5-44, 5-52
  in number expressions 3-341
  in reports 4-29, 6-47
  in time expressions 3-358, 4-19
  searching for NULL 3-49
  with arithmetic operators 3-339, 3-341, 4-19
  with logical operators 3-333, 4-31
  with relational operators 3-334, 4-31
  with string comparisons 3-335, 4-33
  WITHOUT NULL INPUT 5-12
Number expression
  formatting 4-91, 5-42
  syntax and description 3-341
Number of rows processed 2-24
Numeric
  color codes 4-56
  date 3-349
NUMERIC data type 3-313
NUM_ARGS( ) 4-76
NVARCHAR data type E-4, E-6, E-9, E-18

O

Object file 1-33, 1-48, 1-71, B-5
OF keyword
  AFTER GROUP OF control block 6-29
  BEFORE GROUP OF control block 6-31
  DEFINE statement 3-71
  REPORT statement 3-261
  SKIP statement 3-269, 6-52
  TOP OF PAGE clause 6-17
  VARIABLE statement 3-71
OFF keyword, OPTIONS statement 3-231
ON EVERY ROW control block 6-34
ON KEY keywords
  CONSTRUCT statement 3-41
  DISPLAY ARRAY statement 3-87
  INPUT ARRAY statement 3-161
  INPUT statement 3-137, 4-81
  PROMPT statement 3-258
ON keyword
  CONSTRUCT statement 3-35, 3-41
  DISPLAY ARRAY statement 3-87
  INPUT ARRAY statement 3-161
  INPUT statement 3-137
  OPTIONS statement 3-231
  PROMPT statement 3-258
  REPORT statement 3-261
ON LAST ROW block 3-100, 3-261, 6-36
ONCONFIG environment variable D-36
onconfig file, specifying with ONCONFIG D-36
On-line
  files Intro-10
  Help for developers Intro-10
  Help for users 2-21
OPEN FORM statement 3-217
Open NLS environment
  defined E-7, E-19
  mentioned E-4, E-5
OPEN statement
  interrupting 3-236
  USING clause 3-105
OPEN WINDOW statement 3-219
Operands of arithmetic operators 3-320, 3-341, 3-356, 4-21

Operating system
invoking the Compiler from 1-30, 1-59
invoking the Programmers Environment from 1-23, 1-27, 1-52, 1-56
Operators in 4GL statements
associativity and precedence 3-327
built-in operators 4-10
compared with SQL operators 3-330
data types of operands 3-329
field operators 3-328
list of 4-11
query by example 3-49
OPTION keyword
GRANT statement 4-90
MENU statement 3-203
Options of 4GL menus 2-15
OPTIONS statement
mkmessage utility B-2
SQL INTERRUPT 3-64, 3-235, 3-237
syntax 3-228
OR keyword
Boolean operator 3-329, 3-333, 4-31, 4-37
precedence of operators 3-328
OR operator in query by example 3-49
ORDER BY clause
REPORT statement 6-18, 6-31
SELECT statement 6-26
Order of screen fields 3-130, 3-154, 3-232
orders table in stores database A-4
OTHERWISE keyword, CASE statement 3-23
Output
from 4GL programs 6-3
record 3-274, 4-84
Output file
STARTLOG( ) 4-84
UNLOAD statement 3-274
OUTPUT keyword
OUTPUT TO REPORT statement 3-242
REPORT statement 3-260
OUTPUT section of REPORT statement
BOTTOM MARGIN 6-11
LEFT MARGIN 6-12
PAGE LENGTH 6-12
REPORT TO 6-13
RIGHT MARGIN 4-102, 6-14, 6-50
syntax 6-9
TOP MARGIN 6-16
TOP OF PAGE 6-17
OUTPUT TO REPORT statement 3-242, 6-5
Overflow
in a display field 4-91
in data type conversion 3-320, 3-325, C-25, C-29, C-31, C-36
Overriding a Help message 3-37, 3-134, 3-156
Owner naming
CONSTRUCT statement 3-36
DEFINE statement 3-33, 3-69, 3-72, 3-314
in ANSI-compliant database 3-361, 5-19, 5-72
in form specification 5-6, 5-19
INITIALIZE statement 3-126
VALIDATE statement 3-279, 3-361

P

Page eject character 6-17
PAGE HEADER control block 3-271, 4-77, 6-33, 6-37
PAGE keyword
FIRST PAGE HEADER control block 6-33
PAGE HEADER control block 6-37
PAGE LENGTH clause 6-12
PAGE TRAILER control block 6-38
SKIP statement 3-269, 6-52
TOP OF PAGE clause 3-269, 6-17, 6-52
PAGE LENGTH keywords 6-12
PAGE TRAILER control block 6-38
PAGENO operator 3-254, 4-77, 6-38, 6-48
Pages
of a help file message B-3
of a report 3-216, 3-269, 6-10, 6-48, 6-51
of a screen form 5-13, 5-78
of menu options 3-208
of program array records 3-91
of reports 4-77, 4-103
Parameterizing a statement with SQL identifiers 3-251
Parentheses (( )) symbols
Boolean expressions 3-333, 4-31
CHAR data types 3-299
DATETIME values 3-349
function calls 3-328, 4-5
in expressions 3-327
IN operator 3-337, 5-33
in USING format strings 4-92
INTERVAL values 3-307, 3-353
LOAD column list 3-181
SPACE operator 4-82, 6-50
UNITS operator 3-357, 4-21, 4-89
Passing by reference
blob function arguments 3-190, 3-298, 3-317, 4-8, C-6
blob report arguments 3-190, 3-242, 3-298
PATH environment variable D-40
Pathname
including in SQLEXEC D-38
LOAD statement 3-181
specifying with DBPATH D-23
specifying with PATH D-40
UNLOAD statement 3-274
Pattern matching 3-49, 3-336, 4-34
PAUSE statement 3-244, 6-41
P-code runner
customized 1-67
Interactive Debugger 1-56
specifying name and location 1-48
using 1-59
P-code version number 1-60, 1-63, 1-68
Percent (%) symbol wildcard with LIKE 3-336, 4-34

PERCENT(*) aggregate
function 4-14, 6-47
PERFORM (INFORMIX-SQL)
forms with INFORMIX-
4GL 5-77
Period (.) symbol
DATETIME separator 3-351
DECIMAL values 3-342
FLOAT values 3-306
Help message numbers 2-22
in Help files 2-22
in help message source files B-2
in USING format string 4-92
INTERVAL separator 3-355
MONEY values 3-342
prefix separator Intro-10, 3-314,
3-361, 5-19
range operator 3-49
RECORD member 3-328, 3-332
SMALLFLOAT values 3-316
Peripheral device E-23
PICTURE attribute 5-48, 5-49, 5-70
PIPE keyword in REPORT TO
clause 6-13
Planned_Compile option,
PROGRAM Menu 1-21, 1-50
Plus (+) sign
addition operator 3-226, 3-231,
3-328, 3-339, 3-357, 4-20, 4-22
in format strings 4-92
in window border F-6
RECORD declarations 3-72, 3-313
unary operator 3-328, 3-340, 4-92,
C-25
Positioning
a window 3-220
DISPLAY output 3-77
reserved lines 3-226, 3-231
Pound (#) sign
comment indicator 2-6, F-3
in format strings 5-43, 5-48
in USING format strings 4-92
Precedence
in 4GL operators 3-328, 3-329,
4-37
in arithmetic operations 3-339,
4-20
in default values 3-130, 3-154
in display attributes 3-37, 3-292,
5-72
in display elements 4-57
of identifiers 2-13
rules for environment
variables D-6
Precision
DATETIME data type 3-349, 4-42,
4-53
DECIMAL data type 3-68, 3-294,
3-320
FLOAT data type 3-68, 3-294
FORMAT attribute 5-43
in arithmetic operations 3-320
INTERVAL data type 3-353
MONEY data type 3-68, 3-294,
3-312, 3-320
PRECISION keyword 3-305
Predefined identifiers 2-12
PREPARE statement
increasing performance
efficiency 3-253
multi-statement text 3-248, 3-252
parameterizing a statement 3-250
parameterizing for SQL
identifiers 3-251
query by example 3-32
question (?) mark as
placeholder 3-245
restrictions with SELECT 3-246
statement identifier use 3-246
syntax and description 3-245
valid statement text 3-246
variable list 3-315
with DATABASE 3-61
with LOAD 3-181
with UNLOAD 3-274
with . * notation 3-365
Prepared statement
prepared object limit 3-245
valid statement text 3-246
Preprocessor, invoking 1-29, 1-30
Previous key F-6, F-25
PREVIOUS keyword
CONSTRUCT statement 3-44
INPUT ARRAY statement 3-168
INPUT statement 3-142
OPTIONS statement 3-230
Previous Page key
DISPLAY ARRAY statement 3-91
INPUT ARRAY statement 3-162
PREVPAGE keyword 3-234
Print position 3-269
PRINT statement
CLIPPED operator 4-38
in a report 6-42
syntax and description 3-254
USING 4-95
Printable characters 3-345, 5-52
PRINTER keyword
REPORT TO clause 6-13
START REPORT statement 3-272
Printing, specifying print program
with DBPRINT D-26
Privilege
Insert 3-181
Select 3-274
table-level 3-181, 3-274
Procedure 4-5
Process ID 3-188
Program
examples that call C
functions 1-68
flow control statements 3-14
organization statements 3-13
specification database 1-16, 1-45
Program array
ARR_COUNT( ) 4-24
ARR_CURR( ) 4-26
displaying 3-85
SET_COUNT( ) 4-80
PROGRAM attribute 5-28, 5-50,
5-76
Program block
FUNCTION 3-111, 3-112
MAIN 3-191
REPORT 3-260, 6-5
scope of statement labels 3-122,
3-177, 3-284
scope of variables 3-66, 3-261, 6-8
three kinds of 2-7
Program execution
commencing 1-56, 1-63, 4-16, 4-76
from the command line 1-5, 4-16
programs that call C
functions 1-32, 1-64
terminating 4-50

with the Interactive Debugger 1-63, 1-70
Program features
calling C functions 1-64
calling functions 3-16
commenting 2-5
compiler 1-29, 1-36
compiling through Programmers Environment 1-8
compiling, at operating system level 1-58
conditional statements 3-21, 3-287
data validation 3-278
error messages 4-48, B-4
Help messages 4-81
help messages B-2
identifiers 2-9
letter case sensitivity 5-15
multi-module programs 1-11, 1-39
operating system pipes 3-273, 6-13
owner naming 3-361, 5-19
procedural statements 3-13
program arrays 3-85
program blocks 2-7
reports 6-3
running, at operating system level 1-62
screen interaction statements 3-228
screen records 5-64
SQL statements 3-11
suspending execution 3-270
transaction logging 3-104, 3-108, 3-287
types of program modules 1-8, 1-19, 1-26, 1-30, 1-48, 1-67
PROGRAM keyword
EXIT PROGRAM statement 3-98
PROGRAM attribute 3-150, 3-174
Program record
data entry 3-152
declaration 3-72
Programmers Environment
accessing 1-5, 1-6, 1-34
COMPILE FORM Menu 1-14, 1-43
COMPILE MODULE Menu 1-8, 1-37
COMPILE PROGRAM Menu 1-20, 1-49
compiling a form 1-25, 1-54, 5-74
compiling a program 1-10, 1-24, 1-39, 1-53
correcting errors in a program 1-9, 1-38
creating a default form 5-73
Debug option, MODULE Menu 1-40
Debug option, PROGRAM Menu 1-51
defining a program 1-23, 1-52
definition of 1-4
Drop option, PROGRAM Menu 1-22
Exit option, FORM Menu 1-16, 1-45
Exit option, MODULE Menu 1-12, 1-40
Exit option, PROGRAM Menu 1-22, 1-51
files displayed 1-24, 1-62
FORM Menu 1-12, 1-41
Generate option, FORM Menu 1-14, 1-43
in C Compiler version of 4GL 1-6
in Rapid Development System 1-34
INFORMIX-4GL Menu 1-6, 1-35
invoking the Debugger 1-36, 1-51
menu options 1-58
modifying a form specification file 1-13, 1-42
MODULE Menu 1-7, 1-35
NEW FORM Menu 1-15, 1-44
NEW MODULE Menu 1-10, 1-38
NEW PROGRAM Menu 1-20, 1-49
Planned_Compile option, PROGRAM Menu 1-21, 1-50
PROGRAM Menu 1-16, 1-45
Program_Compile option, MODULE Menu 1-11, 1-39
QUERY LANGUAGE Menu 1-22, 1-51
Run option, MODULE Menu 1-11, 1-39
Run option, PROGRAM Menu 1-22, 1-50
Undefine option, PROGRAM Menu 1-51
Program_Compile option, MODULE Menu 1-11, 1-39
Promotable locks 3-105
PROMPT keyword
END PROMPT statement 3-259
OPEN WINDOW statement 3-223
OPTIONS statement 3-230
PROMPT statement 3-255
Prompt line 2-19, 3-94, 3-226, 3-256
PROMPT LINE keywords
OPEN WINDOW statement 3-223
OPTIONS statement 3-230
PROMPT statement
Line mode overlay 3-76
syntax and description 3-255
Prototype
of a function 4-7
of a report 6-6
Pseudo-code 1-4
PSORT_DBTEMP environment variable D-36
Punctuation symbols Intro-7

Q

Qualifiers
database name 3-33, 3-69, 3-361
DATETIME declaration 3-294
DATETIME literals 3-182, 3-302, 3-351
INTERVAL declaration 3-294, 3-353
INTERVAL literals 3-182, 3-309, 3-353
of column names 3-36, 3-69, 3-361
of DATETIME values 4-53, 5-39
of field names 3-359, 3-361, 5-22
of INTERVAL values 5-39
of table names 3-35, 3-69, 3-361, 5-19

owner name 3-33, 3-361
Query by example
CONSTRUCT statement 3-31
range operator 3-49
wildcard characters 3-49
Query optimization
information 3-12
QUERYCLEAR attribute of
PERFORM 5-78
Querying the database
joins 3-362, 5-78
query by example 5-46, 5-47
Question (?) mark
as placeholder in PREPARE 3-245
in WORDWRAP fields 5-62
wildcard with CONSTRUCT 3-49
wildcard with MATCHES 3-335,
4-34
Quit key
with CONSTRUCT 3-41
with DEFER 3-62
with DISPLAY ARRAY 3-92
with INPUT 3-151
with INPUT ARRAY 3-175
with MENU 3-210
with PROMPT 3-259
QUIT keyword, DEFER
statement 3-62
Quit signal 2-23
quit_flag built-in variable 3-41,
3-53, 3-62, 3-199, 3-259
Quotation (") marks
around activation keys 3-199
around character pointer 1-66
around character strings 5-38,
5-45
around database
specification 3-58
around DATETIME literals 3-303
around filenames 3-217, 3-271,
6-13, 6-44
around format strings 4-91, 5-42,
5-48
around INTERVAL literals 3-311
around pipe names 6-13
around SQL identifiers 2-10
around time values 5-38
single and double Intro-7
Quotes
enclosure of monetary values
by E-35, E-48
enclosure of numeric values
by E-48
Quotient 4-22

R

r4gl command 1-5, 1-52, 5-73
r4gldemo script 1-5
Range of values
ASCII characters 3-335, 4-34
COLOR attribute 5-33
DATETIME values 3-49, 3-350
INCLUDE attribute 5-45
INTERVAL values 3-49, 3-354
number expressions 3-341, 4-21,
5-33
query by example 3-49
time expressions 5-33
upscol utility 5-71, B-9
Range test 4-35
Rapid Development System version
of 4GL 1-3, 1-34
REAL data type 3-313
Record
membership (.) operator 3-328,
3-332
SQLCA global record 2-23
RECORD data type 3-19, 3-72,
3-179, 3-313, 3-331
RECORD keyword
data type 3-72, 3-313
defining screen arrays 5-63, 5-66
defining screen records 4-64, 5-64
END RECORD declaration 3-72,
3-313
SCREEN RECORD
specification 3-364
Rectangles in screen forms 4-57
RED attribute 3-290, 5-31, 5-71, F-20
Relational operators 3-48, 3-329,
3-334, 3-337, 3-358, 4-32, 4-36,
4-37
Relay Module
SQLRM environment
variable D-38
SQLRMDIR environment
variable D-39
Release notes Intro-10
Remainder in expressions 3-339,
4-20
Remote database 3-362
Report
aggregates 4-13
driver 3-100, 6-5, 6-39
execution statements 3-113, 3-191
operators 3-328, 3-330, 4-12
writer 6-3
REPORT keyword 3-242
END REPORT statement 6-6
FINISH REPORT
statement 3-100, 6-5
OUTPUT TO REPORT
statement 3-242, 6-5
REPORT statement 6-6
START REPORT statement 3-271,
6-5
REPORT statement
control blocks 6-27
DEFINE section 6-7, 6-8
displaying a report 6-9
FORMAT section 6-7, 6-23
grouping data 6-27
indirect typing 6-8
NEED statement 6-40
ORDER BY section 6-7, 6-18
ORDER EXTERNAL BY 6-18
OUTPUT section 3-272, 6-7, 6-9
passing arguments to 3-242
PAUSE statement 3-244, 6-41
PRINT statement 3-254, 6-42
SKIP statement 3-269, 6-52
statements in a report
definition 6-39
structure 6-7
syntax and description 3-260
with DATABASE 3-61
REPORT TO keywords 6-13
Reports
aggregate functions 6-31
calculations on groups 4-15, 6-47
counting rows 4-14, 6-47
default layout 6-24
features 6-3
formatting 6-23, 6-27

output of a report 6-9
printing output 3-254, 6-42
prototype 6-6
sending output to a file 6-13
sorting data 6-18
REQUIRED attribute 5-28, 5-52
Reserve size of VARCHAR data type 3-68, 3-294
Reserved lines
clearing 3-27
Comment line 2-19, 3-51, 3-80, 3-226, 5-36
default locations 2-19, 3-94, 3-226
Error line 2-19, 3-51, 3-80, 3-96, 3-227, 4-49, 4-50
Form line 2-19, 3-93, 3-226
in current window 3-56
Menu help line 2-19
Menu line 2-17, 3-193, 3-226
Message line 2-19, 3-213, 3-226
positioning 3-226, 3-231, 4-57
Prompt line 2-19, 3-226, 3-256
Reserved values 3-306
Reserved words
as identifiers 2-10
listing H-1
Resume option
Help menu 2-21
Help window 4-81
RETURN character in WORDWRAP reports 6-51
Return key
in ON KEY clause 3-41
in query by example 3-52
RETURN keyword
OPTIONS statement 3-234
RETURN statement 3-19, 3-263
RETURN statement 3-19, 3-263
RETURNING keyword
CALL statement 3-19
RUN statement 3-267
REVERSE attribute 3-97, 3-290, 3-291, 5-28, 5-31, 5-53, 5-71, F-10
RIGHT keyword
attribute of PERFORM 5-78
OPTIONS statement 3-234
RIGHT MARGIN keywords
OUTPUT section 6-14
PRINT statement 3-254
WORDWRAP operator 4-102, 6-50
Ring menu 2-15, 2-16
ROLLBACK WORK statement
interrupting transactions 3-237, 3-239
with LOAD 3-185
with SQLEXIT( ) 4-83
with WHENEVER 3-285
Rounding error 3-305, 3-320, 3-325, 3-343, 5-43
ROW keyword
EVERY ROW statement 6-24
INPUT ARRAY statement 3-160, 3-166
ON EVERY ROW control block 6-34
ON LAST ROW control block 6-36
REPORT statement 3-261
ROWID keyword 2-24, 3-181, 3-330
ROWS keyword in OPEN WINDOW statement 3-221
Run option
MODULE Menu 1-11, 1-39
PROGRAM Menu 1-22, 1-50
RUN statement 3-265
Runner
command to invoke 1-58
creating a customized 1-67
specifying location of 1-48
using to execute p-code 1-4
Running a 4GL program
command line 1-5, 1-32
that calls C functions 1-32, 1-70
using Debugger 1-58
Run-time
errors, untrappable 3-283
program, setting DBANSIWARN D-8

S

Scale
DATETIME data type 3-349, 4-42
DECIMAL data type 3-68, 3-294, 3-320, C-39
FORMAT attribute 5-43
INTERVAL data type 3-353
MONEY data type 3-68, 3-294, 3-312
Scope of reference
4GL identifiers 2-11
4GL windows 3-219
global variables 2-11, 3-117
identifiers of form entities 2-11
program variables 2-11, 2-13, 3-192
screen array 5-67
screen form 2-11, 3-29
screen record 5-65
SQL identifiers 2-12
Screen
interaction statements 3-14
menu option 4-81
option of Help menu 2-21
Screen array
binding to program records 3-153
clearing 3-27
cursor movement 3-172
format of 5-16
identifying the current row 4-26, 4-78
in field clause 3-359
scrolling 3-172, 3-268
syntax 5-66
testing with FIELD_TOUCHED( ) operator 4-64
Screen display characteristics
clearing the screen F-5, F-24
default screen attributes B-8
Screen form
closing 3-218
current 3-128, 3-152
specifying from the Programmers Environment 1-12, 1-41, 5-73
SCREEN keyword
CLEAR SCREEN statement 3-27
CLEAR WINDOW SCREEN statement 3-27
CURRENT WINDOW statement 3-56
INSTRUCTIONS section 5-7
referencing the default window 2-19
SCREEN section 5-12

Screen record
clearing 3-27
default screen record 5-64
in field clause 3-359, 4-66
order of components 3-36, 3-154, 3-364, 5-65
scope of reference 5-65
within a screen array 3-85, 3-268, 5-67
SCREEN RECORD keywords 4-64, 5-63, 5-64, 5-66
SCREEN section of form specification
display field 5-14
field delimiters 5-69
field labels 5-15
field length 5-76
field tags 5-14, 5-20, 5-76
graphics characters 5-16
screen layout 5-14
syntax 5-12
SCROLL keyword
DECLARE statement 3-105
SCROLL statement 3-268
SCROLL statement 3-268
Scrolling keys 3-172
SCR_LINE( )
function 4-78
with DISPLAY ARRAY 3-90
with INPUT ARRAY 3-171
SECOND keyword
DATETIME qualifier 3-301, 3-349, 4-42
INTERVAL qualifier 3-307, 3-353
UNITS operator 4-89
SELECT keyword
GRANT statement 3-274
INSERT statement 3-184
SELECT statement 4-9
Select privilege 3-274
SELECT statement
copying rows to an ASCII file 3-274
displaying results 6-3
interrupting 3-236
query by example 3-32
requiring no cursor 4-9
restrictions with INTO clause 3-246
Semicolon (;) symbol
as a statement terminator 2-5
in a field description 5-21
in PRINT statements 6-37, 6-45
SERIAL data type
as INTEGER variables 3-69, 3-307
display fields 5-47
in input files 3-182
in program records 3-314
in UPDATE statement 3-365
INSERT statement 5-47
SQLCA.SQLERRD[2] 2-24
Set membership test 3-50, 3-337, 4-35
Setting environment variables D-4
SET_COUNT( )
function 4-80
with DISPLAY ARRAY 3-85, 3-90
with INPUT ARRAY 3-171
sg1 terminal specification F-4, F-19
Shaded syntax diagram elements Intro-7
Shared memory parameters, specifying file with ONCONFIG D-36
Shell
setting environment variables in a file D-2
specifying with DBREMOTECMD D-27
SHOW keyword, MENU statement 3-204
showhelp function B-3
SHOWHELP( )
function 4-81
ON KEY clause 2-21, 3-37, 3-134, 3-156
Signals
Interrupt 3-62
Quit 3-62, 3-237
SIGQUIT signal 3-237
Simple data type 3-68, 3-294
Single-character fields 5-76
Single-precision floating-point number, storage of 3-305
SIZE keyword, form specification 5-13
SKIP statement 3-269, 6-52
Slash (/) symbol
database specification 3-58
DATE literals 3-300, 3-321
division operator 3-328, 3-339, 3-357, 4-20, 4-22
SLEEP statement 3-270
SMALLFLOAT data type
data type conversion 3-319, 3-324
declaration 3-68, 3-294
description 3-315
display width 3-78, 5-76, 6-44
FORMAT attribute 5-43
in NLS E-9
literal values 3-316, 3-342
SMALLINT data type
conversion 3-319
data type conversion 3-324
declaration 3-68, 3-294
description 3-316
display width 3-78, 5-76, 6-44
in report output 6-44
literal values 3-342
SOME keyword in SQL Boolean operator 3-330
Sorting data
in a report 6-18
with a cursor 6-22
Sorting, PSORT_DBTEMP environment variable D-36
Source
compiler 3-280
modules 1-29, 1-58, 1-60, 2-7
path 1-47
SPACE or SPACES operator 3-328, 4-82, 6-49
Spacebar 3-209
SPL expressions 3-330
SQL
built-in functions 4-5, 4-6
INTERRUPT option 3-64, 3-235, 3-237
keyword in OPTIONS statement 3-231
version number 1-31, 1-60, 1-63, 1-68
SQL language
accessing from the Programmers Environment 1-22, 1-51
concurrency control 3-105

cursor manipulation statements 3-12
data access statements 3-12
data definition statements 3-12
data integrity statements 3-13
data manipulation statements 2-12, 3-12
expressions 3-330
interactive query language 1-7, 1-35, 5-78
interrupting statements 3-64, 3-235, 3-237
operators 3-330
query optimization statements 3-12
testing statement execution 2-23
transaction logging 3-108, 3-185
types of statements 2-5
views 3-69

SQLAWARN
- characters 2-25
- global record 3-61, 3-284, 3-320

SQLCA record
- definition of 2-23
- effect of setting DBANSIWARN D-8
- SQLAWARN 2-25
- SQLCODE 2-24, 3-236
- SQLERRD 2-24
- WHENEVER ERROR condition 3-282

SQLCODE global variable 3-236, 3-282
SQLERROR keyword 3-281, 3-283
SQLEXEC environment variable D-38
SQLEXIT( ) 4-83
SQLRM environment variable D-38
SQLRMDIR environment variable D-39
SQLWARNING keyword 3-284
Stack argument 2-27, C-2
START REPORT statement 3-271, 6-5
STARTLOG( ) 4-84
state table in stores database A-7

Statement
- blocks 2-8, 3-102
- labels 2-7, 3-114, 3-122, 3-177, 3-281
- terminator 2-5

Statement identifier
- definition of 3-246
- releasing 3-246
- syntax
  - in PREPARE 3-245
- use
  - in PREPARE 3-246

Statement segments
- asterisk (*) notation 3-363
- ATTRIBUTE clause 3-290
- data types of 4GL 3-293
- expressions of 4GL 3-326
- field clause 3-359
- notational conventions Intro-7
- table qualifiers 3-361
- THRU or THROUGH keywords 3-363

Statement syntax
- CALL 3-16
- CASE 3-21
- CLEAR 3-21, 3-26
- CLOSE FORM 3-29
- CLOSE WINDOW 3-30
- CONSTRUCT 3-31
- CONTINUE 3-55
- CURRENT WINDOW 3-56
- DATABASE 3-58
- DEFER 3-62
- DEFINE 3-65
- DISPLAY 3-74
- DISPLAY ARRAY 3-85
- DISPLAY FORM 3-93
- END 3-95
- ERROR 3-96
- EXIT 3-98
- FINISH REPORT 3-100
- FOR 3-102
- FOREACH 3-105
- FUNCTION 3-111
- GLOBALS 3-117
- GOTO 3-122
- IF 3-124
- INITIALIZE 3-125
- INPUT 3-128
- INPUT ARRAY 3-152
- LABEL 3-177
- LET 3-178
- LOAD 3-181
- LOCATE 3-186
- MAIN 3-191
- MENU 3-193
- MESSAGE 3-213
- NEED 3-216, 6-40
- OPEN FORM 3-217
- OPEN WINDOW 3-219
- OPTIONS 3-228
- OUTPUT TO REPORT 3-242
- PAUSE 3-244, 6-41
- PRINT 3-254, 6-42
- PROMPT 3-255
- REPORT 3-260, 6-6
- RETURN 3-263
- RUN 3-265
- SCROLL 3-268
- SKIP 3-269, 6-52
- SLEEP 3-270
- START REPORT 3-271
- UNLOAD 3-274
- VALIDATE 3-278
- WHENEVER 3-281
- WHILE 3-287

Statement type
- compiler directive 3-14
- cursor manipulation 3-12
- data access 3-12
- data definition 3-12
- data integrity 3-13
- data manipulation 3-12
- definition and declaration 3-13
- program flow control 3-14
- query optimization 3-12
- report execution 3-14
- screen interaction 3-14
- storage manipulation 3-14

Statements in reports
- NEED 3-216
- PAUSE 3-244, 6-41
- PRINT 3-254
- SKIP 3-269

Status code
- after data type conversion C-25
- after database disconnection 4-83
- after program termination 3-99
- of a child process 3-265

status variable
definition of 2-24
interrupting SQL
statements 3-236
set to 100 3-105
VALIDATE statement 3-279
WHENEVER statement 3-283
with ERR_GET( ) 4-48, 4-49
with ERR_PRINT( ) 4-49
with ERR_QUIT( ) 4-50
STEP keyword, FOR
statement 3-103
stock table in stores database A-5
STOP keyword in WHENEVER
statement 3-285
Storage manipulation
statements 3-14
stores database
creating A-3
customer table A-3
customer table columns A-3
data values A-14
items table A-4
items table columns A-4
join columns A-8, A-8 to A-14
manufact table A-6
manufact table columns A-6
map of A-8
orders table A-4
orders table columns A-4
overview A-1
restoring the original A-2
state table A-7
state table columns A-7
stock table A-5
structure of tables A-3, A-8
tables in A-3
stores2 database Intro-12
String comparison 3-329, 3-335,
4-33, 4-37
String value
NULL 5-25
substring 5-22
Structure definition file,
function 1-64
Structured
data types 3-70, 3-296
programming 4-5
stty -istrip command E-30
Subdiagram
box symbol Intro-7
graphic notation Intro-7
Subroutine 4-5
Subscript
of a character column 5-22
to specify array elements 3-297
to specify substrings 3-317
Substring
in a screen field 5-22
of character array elements 3-297
of character variables 3-179, 3-215
of TEXT values 3-317
Subtraction (-) operator
number expressions 3-339, 4-22
precedence 3-328
precision and scale 3-321
reserved lines 3-226, 3-231
returned values 4-20
time expressions 3-357, 4-22
SUM( ) aggregate function 2-25,
4-14, 4-95, 6-46
Syntax diagram
conventions Intro-6
elements of Intro-8
Syntax of command line to compile
a 4GL source file 1-30, 1-59
syscolatt table
changing color names F-20
color and intensity values 5-70,
B-8
creating 5-72
creating with upscol B-5
DISPLAY LIKE attribute 5-40
FGL_DRAWBOX( )
arguments 4-56
INPUT ARRAY statement 3-155
INPUT statement 3-133
precedence of attributes 5-72
schema 5-69
with FORM4GL 5-21, 5-69
syscolumns table 3-69, 3-184, 3-363
syscolval table
as used by INITIALIZE 5-71
creating 5-72
creating with upscol B-5
data validation 5-71
INITIALIZE statement 3-126,
3-127
INPUT ARRAY statement 3-154
INPUT statement 3-130
schema 5-69
VALIDATE LIKE attribute 5-55
VALIDATE statement 3-280
with FORM4GL 5-21, 5-69
syspgm4gl files 1-16, 1-45
systables table 4-9
System catalog
syscolumns 3-69
systabauth 4-90
systables 4-9, 5-18
System clock
CURRENT operator 4-42
DATE operator 4-44
EXTEND( ) operator 4-54
TIME operator 4-86
TODAY operator 4-87

T

TAB character
in report output 4-103, 6-51
in statements 2-3
in TEXT values 3-317, 3-345
TAB key
in ON KEY clause 3-41
in query by example 3-52
order of fields 3-50, 3-232
reassigning its function 3-41,
3-234
Table
alias for table name 5-19
changing column data
types 3-319
current 5-78
inserting data 3-181
joining tables A-8
locking 3-185
qualifiers 3-35, 3-361
reference 3-359, 5-20
structure in stores database A-3
temporary 3-101, 4-13, 6-22
Table alias
declaring 5-19
naming conventions 2-10
qualifiers 3-361

TABLE keyword in LOCK TABLE statement 3-185
TABLES section of form specification
  description 5-6
  syntax 5-18
Temporary
  files, specifying directory with DBTEMP D-29
  tables, specifying dbspace with DBSPACETEMP D-28
TERM environment variable D-41
TERMCAP environment variable D-41
termcap file
  and TERMCAP environment variable D-41
  description F-2
  graphics characters 5-17
  selecting with INFORMIXTERM D-35
Terminal bell, ringing 3-96
Terminal handling
  and TERM environment variable D-41
  and TERMCAP environment variable D-41
  and TERMINFO environment variable D-42
Termination status 3-99, 3-266
Terminator, in syntax diagrams Intro-7
terminfo directory
  and TERMINFO environment variable D-42
  selecting with INFORMIXTERM D-35
TERMINFO environment variable D-42, F-20, F-26
terminfo files 5-17, F-20
Text cursor
  in a field 5-5
  in disabled fields 5-6
  with CONSTRUCT 3-32
  with DISPLAY ARRAY 3-86
  with INPUT 3-130, 3-146
  with INPUT ARRAY 3-172
  with MENU 3-199
TEXT data type
  Boolean expressions 4-33, 5-33
  data entry 3-150, 3-174
  declaration 3-65, 3-70
  description 3-317
  display fields 3-149, 5-50, 5-57
  display width 5-76, 6-44
  in expressions 3-332, 3-343
  in input files 3-182
  in output files 3-275
  in program records 3-72, 3-313
  in report output 4-102, 6-44, 6-50
  initializing 3-186
  large data type 3-296
  passing by reference 3-18, 3-264
  query by example 3-48
  selecting a TEXT column 3-317
  storing control characters 3-317
  syscolval table 3-280
  unprintable characters 3-317, 3-345
Text editor 1-9, 1-13, 1-29, 1-38, 1-42, 1-58, D-11
THEN keyword, IF statement 3-124
Thousands separator D-14, D-23, E-31, E-33, E-35, E-40, E-45
THROUGH keyword 3-65, 3-125, 3-314, 3-363, 5-65
THRU keyword 3-65, 3-81, 3-125, 3-133, 3-314, 3-363, 5-65
Time data types 3-295, 3-347
Time expressions
  as operands 4-21
  formatting 4-91
  syntax 3-348
TIME operator 3-330, 4-86
Time units
  in data type conversion 3-321
  in DATETIME qualifiers 3-301, 3-350
  in INTERVAL qualifiers 3-307, 3-353, 5-39
  in numeric dates 3-300, 3-349, 4-44, 4-94
  with EXTEND( ) operator 4-53
  with MDY( ) operator 4-74
Title of a menu 2-15
TO keyword
  DATETIME qualifier 3-301, 3-350, 4-42, 5-39
  DISPLAY ARRAY statement 3-85
  DISPLAY statement 3-74, 3-364
  EXTEND( ) operator 4-53
  FOR statement 3-102
  INCLUDE attribute 5-44
  INITIALIZE statement 3-127, 3-364
  INTERVAL qualifier 3-307, 3-353, 5-39
  OUTPUT TO REPORT statement 3-242, 6-5
  REPORT TO clause 3-271, 6-13
  SKIP statement 3-269, 6-52
  START REPORT statement 3-271
  UNLOAD statement 3-274
  WHENEVER statement 3-177, 3-281
TODAY operator 4-87, 5-39
TOP MARGIN keywords 6-16, 6-34
TOP OF PAGE clause
  OUTPUT section 6-18
  SKIP statement 3-269, 6-52
TRAILER keyword, REPORT statement 6-38
Trailing blank spaces
  CLIPPED operator 4-38
  VARCHAR values 3-318
Trailing currency symbol D-14, D-21, E-33, E-40, E-45
Transaction logging
  explicit transactions 3-237, 3-238
  For loops 3-104
  FOREACH statement block 3-108
  implicit transactions 3-237, 3-238
  interrupting SQL statements 3-237
  singleton transactions 3-237, 3-238
  SQLEXIT( ) function 4-83
  while loading data 3-185
  WHILE loop 3-287
TRUE (Boolean constant) 3-62, 3-288, 3-333, 5-31
Truncation of data 2-25, 3-86, 3-182, 3-275, 3-325, 4-22, 4-58, 4-103, 5-59, 6-51

Two-pass report 3-100, 3-101, 3-262, 4-14, 6-22
TYPE keyword in FORMONLY fields 5-24, 5-45
Types of statements
  4GL statements 3-13
  SQL statements 3-11
Typographical conventions Intro-5
Typover editing mode 3-52, 3-147, 3-173

U

Unary minus (-) symbol 2-6, 3-328, 3-340, 3-342, 3-348, 4-19, 4-92, 5-39
Unary plus (+) symbol 3-316, 3-328, 3-340, 3-342, 3-348, 4-19, 4-92
UNCONSTRAINED keyword in OPTIONS statement 3-51, 3-230, 3-232
Undefine option, PROGRAM Menu 1-51
Underflow conversion error C-25, C-36
UNDERLINE attribute 3-290, 3-291, 5-31, 5-71, F-10
Underscore (_) symbol
  in field tags 5-15
  in identifiers 2-10
  wildcard with LIKE 3-336, 4-34
Units of time
  CURRENT operator 4-42
  DATE operator 4-44
  DATE values 3-300, 3-321, 3-349
  DATETIME values 3-303, 3-321, 3-351
  DAY( ) operator 4-46
  EXTEND( ) operator 4-54
  INTERVAL values 3-307, 3-355
  MDY( ) operator 4-74
  UNITS operator 4-89
  YEAR( ) operator 4-104
UNITS operator
  data type conversion 3-339
  in arithmetic expressions 3-357, 4-21, 4-89
  precedence as operator 3-328
  specifying a default value in a field 5-39
  syntax and description 4-89
UNIX
  default print capability in BSD D-5, D-26
  default print capability in System V D-5, D-26
  environment variable setting in BSD and System V D-4
  environment variables listed D-7
  terminfo library support in System V D-35
  viewing environment settings in BSD D-4
  viewing environment settings in System V D-4
UNLOAD statement
  in NLS E-7, E-10, E-15, E-35, E-40, E-42
  interrupting 3-236
  specifying field delimiter with DBDELIMITER D-11
  syntax and description 3-274
Unprintable characters 3-345, 4-103
Unquoted literal 5-39
Unsigned values 3-339, 4-20
Untrappable errors 2-26, 3-97, 3-283
UP keyword
  OPTIONS statement 3-234
  SCROLL statement 3-268
  syscolval table 5-70, B-7
Updatable views 5-29
UPDATE
  keyword in DECLARE statement 3-105
  statement, interrupting 3-236
  SYSCOL Menu (upscol) B-5
UPDATE STATISTICS statement and DBUPSPACE environment variable D-29
Uppercase characters
  DEFAULT attribute 5-39
  DOWNSHIFT attribute 5-41, B-7
  DOWNSHIFT( ) 4-47
  in field tags 5-15
  in identifiers 2-4, 2-10
  INCLUDE attribute 5-45
  SHIFT attribute 5-70, B-7
  typographic convention Intro-5
  UPSHIFT attribute 5-54, B-7
  UPSHIFT( ) 4-90
upscol utility 3-126, 3-280, 5-27, 5-40, 5-69, 5-72, B-5
UPSHIFT attribute 5-28, 5-54, 5-69, 5-70, B-7, E-10
UPSHIFT( ) 4-90
USER keyword 3-330, 4-9
User locale E-2
USING expression
  in NLS E-10, E-31, E-33, E-39, E-40, E-42, E-44
  NLS example E-34, E-46
USING keyword
  OPEN statement 3-105
  USING operator 4-91
USING operator
  DISPLAY statement 3-78
  MESSAGE statement 3-213
  PRINT statement 4-94, 6-44
  syntax and description 4-91
Utility programs
  mkmessage B-2
  upscol 5-27, 5-69, B-5

V

V command-line option 1-31, 1-60, 1-63, 1-68
VALIDATE LIKE attribute 5-18, 5-28, 5-55
VALIDATE Menu (upscol) B-7
VALIDATE statement 3-278, 3-283, 5-71
Validation errors 2-26, 3-279, 3-283
VALUES keyword in INSERT statement 3-184, 5-4
VARCHAR data type
  data type conversion 3-324
  declaration 3-68, 3-294
  description 3-318
  display fields 5-48, 5-57
  display width 3-78, 5-76, 6-44
  in expressions 3-343
  in input files 3-182

in NLS E-4, E-6, E-9, E-18
in output files 3-275
in report output 6-44
pattern matching 3-336, 4-34
substrings 5-22
unprintable characters 3-345
Variables
allocating 3-186
as operands 3-331
binding to database columns 5-3
declaring 3-65, 3-113, 3-192, 6-8
global 2-11, 3-66, 3-117
implicit names 3-69, 3-72
in DATABASE statement 3-59
in REPORT statement 6-7
indirect typing 3-59, 3-69, 6-8
local 2-11, 3-114, 3-192, 3-261
maximum number with p-code compiler 3-71
modular 2-11, 3-66
naming rules 2-10
scope of reference 2-11, 3-114, 3-192
status variable 2-24
visibility 2-13, 3-117
VERIFY attribute 5-28, 5-56, 5-70
Version numbers of SQL software 1-31, 1-60, 1-63, 1-68
Versions of 4GL 1-3
Vertical (|) bar
default delimiter 3-184, 3-276
field separator in forms 5-68
graphics character 5-16, F-6, F-26
in termcap specifications F-3
in window border F-6
OR symbol with CONSTRUCT 3-49
SQL concatenation operator 3-330
Video attributes 2-19, 3-290, 5-27
View
in form specification file 5-29
in FROM clause of CONSTRUCT 3-36
in INSERT clause of LOAD 3-181
in LIKE clause of DEFINE 3-69
Visibility of identifiers 2-13, 3-65, 3-114, 3-117

W

W warning character in SQLAWARN 2-25, 3-61
WAITING keyword, RUN statement 3-267
Warning
conditions 2-25, 3-61, 3-281, 3-320
messages 1-31
WARNING keyword in WHENEVER statement 2-25, 3-281
WEEKDAY( ) operator 4-100
WHEN keyword, CASE statement 3-22
WHENEVER statement
syntax and description 3-281
trapping errors B-4
versus GOTO statement 3-122
with ERROR statement 3-97
with ERRORLOG( ) 4-51
with LABEL statement 3-177
with STARTLOG( ) 4-85
WHERE clause
aggregate functions 4-14, 6-46
pattern matching 3-49
query by example 3-32, 3-49
with COLOR attribute 5-32, 5-78, B-9
WHERE keyword
COLOR attribute 3-330, 5-31
Debugger command 3-99
SELECT statement 3-34, 3-275, 3-330
WHILE keyword in CONTINUE WHILE statement 3-55
WHILE statement 3-287
WHITE attribute 3-290, 5-31, 5-71, F-20
Wildcard symbols
CONSTRUCT statement 3-49
in syscolatt table 5-71
with LIKE 3-336, 4-34, 4-101
with MATCHES 3-335, 4-34
Window
border F-6, F-25
clearing 3-26
closing 3-30
current 2-19
display attributes 3-292, 5-72
naming conventions 2-10
opening 3-219
reserved lines 2-19, 3-228
stack 2-20, 3-30, 3-56, 3-220
WINDOW keyword
CLEAR WINDOW statement 3-27
CURRENT WINDOW statement 3-56
OPTIONS statement 3-234
WITH FORM clause in OPEN WINDOW statement 3-222
WITH keyword
DECLARE statement 3-105
GRANT statement 4-90
OPEN WINDOW statement 3-29, 3-221
WITHOUT DEFAULTS keywords
INPUT ARRAY statement 3-154, 5-38
INPUT statement 3-131, 5-38
with SET_COUNT( ) 4-80
WITHOUT keyword
INPUT ARRAY statement 3-154
INPUT statement 3-131
RUN statement 3-267
WITHOUT NULL INPUT keywords in DATABASE section 3-130, 3-154, 5-10, 5-38
WORDWRAP keyword
CONSTRUCT statement 3-51
INPUT statement 3-148
PRINT statement 3-254, 6-50
WORDWRAP attribute 5-26, 5-28, 5-57
WORDWRAP operator 3-328, 4-102
WORK keyword
BEGIN WORK statement 3-285
COMMIT WORK statement 3-185
ROLLBACK WORK statement 3-185, 3-285
WRAP keyword in OPTIONS statement 3-52, 3-230

X

X symbol in format strings 5-48
xmc1 terminal specification F-22
XOFF key 3-88, 3-139, 3-163, 3-235, 3-259
XON key 3-88, 3-139, 3-163, 3-235, 3-259
XPG3 E-2, E-3
X/Open-defined environment variable E-11, E-12

Y

y symbol
  in format strings 4-94, 5-42
  values in syscolatt table 5-70
Y symbol values in syscolatt table B-8
YEAR keyword
  CURRENT operator 4-42
  DATETIME qualifier 3-301, 3-349, 4-42, 5-25
  EXTEND operator 4-54
  INTERVAL qualifier 3-309, 3-353, 5-39
  UNITS operator 4-89
  YEAR( ) operator 4-104
YEAR( ) operator 4-104
YELLOW attribute 3-290, 5-31, 5-71, F-20
YES
  keyword in syscolval table B-7
  value in syscolval table 5-70

Z

ZA function (termcap file) F-10, F-18
Zero
  as divisor 3-339, 4-22, C-36
  as MOD operand 4-22
  byte (ASCII 0) 3-346, 6-47
  DATE to DATETIME conversion 3-321
  default INTERVAL value 3-131, 3-154, 3-323, 5-12, 5-38
  default MONEY value 5-38
  default number value 3-131, 3-154, 5-25, 5-38
  entering SERIAL values 3-182
  in Boolean expressions 3-333, 4-30, 5-34
  in output files 3-275
  or more characters, symbol for 3-49, 3-336, 4-34
  preserving leading zeros 3-299
  scale in arithmetic 3-320
  status code of SQL 2-24, 3-282, 3-288
  status returned by SQLEXIT( ) 4-83
  WEEKDAY( ) value 4-100
  zero fill (&) character 4-93
ZEROFILL attribute of PERFORM 5-78

Symbols

?, question mark as placeholder in PREPARE 3-245