INFORMIX®-4GL

Concepts and Use

Version 6.0
March 1994
Part No. 000-7610

Published by INFORMIX® Press

Informix Software, Inc.
4100 Bohannon Drive
Menlo Park, CA 94025

ACKNOWLEDGMENTS

The following people contributed to this version of *INFORMIX-4GL Concepts and Use*:
Kaye Bonney, Diana Boyd, Lisa Braz, Patrick Brown, Tom Houston, Todd Katz, and Liz Knittel

Preface

This book describes **INFORMIX-4GL** at three levels:

What?	Part I covers the main features of **4GL**, describes the kind of work it is meant to do, and the ways it is normally used. Read Part I for a conceptual overview.
Why?	Part II covers the fundamental ideas behind the design of **4GL** so you will know its parts and how they fit together. Read Part II to understand **4GL** as a programming language and for orientation before you begin using it.
How?	Part III explores **4GL** in depth, using examples and discussion to show how its statements are used together to build an application. Read Part III to gain a framework that links together the very detailed topics of the *INFORMIX-4GL Reference*.

This book has a companion, the *INFORMIX-4GL Reference*. It shows every part of the language in great detail. This book does not cover every feature of every statement, but after reading it you will have the vocabulary you need to understand the topics in the *INFORMIX-4GL Reference*, and you will know which topics of that book you need to read for any situation.

To run your 4GL programs, you also must have access to a database server (**INFORMIX-OnLine Dynamic Server** or INFORMIX-SE). The database server either must be installed on your machine or on another machine to which your machine is connected over a network.

Summary of Chapters

INFORMIX-4GL Concepts and Use is divided into these parts and chapters:

- This Preface provides general information about this book and lists additional reference materials that might contribute to your understanding of relational database management and program design.
- The Introduction tells how **4GL** fits into the Informix family of products and manuals, style conventions in the manual, and information about useful on-line files.

Part I – Overview

Chapters 1 through 3 contain a survey of the language at a high level. It describes the most important language features and the kinds of applications it is meant to build.

- Chapter 1, "Introducing INFORMIX-4GL," provides some basic information on **4GL**, what it includes, what it is used for, and where the program can be run.
- Chapter 2, "Interfaces of INFORMIX-4GL," considers **4GL** as a tool that provides the necessary connectivity so that interactive forms, SQL databases, sequential files, and reports can work together to provide users with the ability to rapidly access and modify information.
- Chapter 3, "The INFORMIX-4GL Language," introduces the structured procedural and non-procedural aspects of the language.

Part II – Concepts

Chapters 4 through 6 cover the ideas that are basic to the design of the language. It emphasizes the programming tasks and problems that **4GL** is meant to solve and how the features of the language are designed to solve them. Some partial coding examples are used to illustrate the main points.

- Chapter 4, "Parts of an Application," offers an overview of the components of a **4GL** application.
- Chapter 5, "The Procedural Language," describes three basic **4GL** language features: data definition, decisions and loops, and handling error conditions.
- Chapter 6, "Database Access and Reports," examines the relationship between the data in a SQL database and **4GL** reports.

Part III – Usage

Chapters 7 through 12 are more specific, covering the statements of the language in groups, discussing how specific statements work with others to solve common programming tasks. Many short samples of code are used for illustration.

- Chapter 7, "The User Interface," reviews the major components of an interactive **4GL** application.
- Chapter 8, "Using the Language," details the data types available in **4GL**, variables, data structures and use of arrays, and other, similar topics.
- Chapter 9, "Using Database Cursors," overviews nonprocedural, row-by-row, and dynamic methods of accessing an SQL database from a **4GL** application.
- Chapter 10, "Creating Reports," shows how to design **4GL** report drivers and report formatters.
- Chapter 11, "Using the Screen and Keyboard," takes a detailed look at the methods of specifying a screen form and managing **4GL** windows.
- Chapter 12, "Handling Exceptions," looks at the problem of handling anticipated and unanticipated situations when running a **4GL** application.

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)

Table of Contents

INFORMIX-4GL Concepts and Use

Index

Introduction

INFORMIX-4GL consists of a suite of tools that allow you to efficiently produce complex interactive database applications. Using the 4GL language, you can quickly write sophisticated, portable, forms-driven, full-screen applications for data entry, data lookup and display, and report generation.

The 4GL development environment provides all the tools necessary to design screen forms, construct and manage program modules, and compile source modules.

Documentation Included with 4GL

The 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.

Manual	Description
INFORMIX-4GL Quick Syntax	Contains the syntax diagrams from the *INFORMIX-4GL Reference*, the *Guide to the INFORMIX-4GL Interactive Debugger*, and the *Informix Guide to SQL: Syntax*, Version 6.0.
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.
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 Administrator's Guide*, Version 6.0 or the *INFORMIX-SE Administrator's Guide*, Version 6.0.

How to Use this Manual

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

The following section describes the typographical conventions used in this manual.

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.`

Additionally, when you are instructed to "enter" or "execute" text, immediately press RETURN after the entry. When you are instructed to "type" the text or "press" a key, pressing RETURN is not 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 Reference

Use the **finderr** script to display a particular error message or messages on your terminal screen. Use this script by typing `finderr` *msg_num* where *msg_num* is the number of the message you want to look up.

The script is located in the **$INFORMIXDIR/bin** directory. For details on using this script, see the introduction of *INFORMIX-4GL Reference*.

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 of *INFORMIX-4GL Reference*.

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 geared towards 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. 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 of *INFORMIX-4GL Reference*.

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 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.

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

Introducing INFORMIX-4GL

Overview

This chapter contains a high-level overview of **INFORMIX-4GL**. Its aim is to orient you to the capabilities and typical uses of the product, and to answer general questions such as what kind of software **4GL** is and what is it used for.

What is 4GL?

4GL is a full-featured, general-purpose, fourth-generation programming language with special facilities for producing the following:

- Database query and database management using the Structured Query Language (SQL).
- Reports from a database or other data.
- Form- and menu-based multi-user applications.

These special features make **4GL** especially well-suited to developing large database applications.

4GL Provides a Programmer's Environment

4GL provides a *Programmer's Environment* that makes it easy to create, compile, and maintain large, multi-module programs. Within the Programmer's Environment, you can:

- Create new program modules and modify existing modules.
- Compile individual modules and entire applications.
- Create and compile forms used by the application.
- Run compiled applications.
- Use INFORMIX-SQL to interact with an Informix database.
- Get help at any time by using the on-line help feature.

You can also manage your applications by using commands at the operating system prompt rather than using the Programmer's Environment.

4GL Works with Databases

Although it is a complete, general-purpose programming language, **4GL** was specifically designed to make certain kinds of programs especially easy to write. Programs of these kinds, collectively "interactive database applications," face some or all of the following special challenges:

- They retrieve data from a local or remote database and process it with logic more complicated than SQL alone permits.
- They present data using screen forms and allow users to construct queries against the database.
- They allow users to alter database records, often enforcing complex requirements for data validation and security.
- They update a database with data processed from other databases or from operating system files.
- They generate multi-page, multi-level reports based on data from a database or other sources, often letting the user set the parameters of the report and select the data for it.

With **INFORMIX-4GL** you can program applications of these kinds more easily than with any other language.

In addition, **4GL** has an open, readable syntax that encourages good individual or group programming style. Programs written in **4GL** are easily enhanced and extended. This, with its development environment, makes it easy for programmers to become productive quickly no matter what programming languages they know.

4GL Runs in Different UNIX Environments

INFORMIX-4GL is the only multi-purpose programming language that offers code- and display-compatibility across operating environments. Applications you develop are portable to the different platforms subject to simple porting guidelines.

You can run this version of **4GL** on the following types of computers:

- On UNIX character-based terminals provided by a wide variety of hardware vendors
- On UNIX workstations

Informix also provides a Microsoft Windows version of **4GL**.

Two Versions of 4GL

Informix provides two versions of **4GL**:

- 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. (This version is sometimes abbreviated as **RDS**.)

For details on the differences between the two versions, see Chapter 1 in *INFORMIX-4GL Reference*.

Chapter

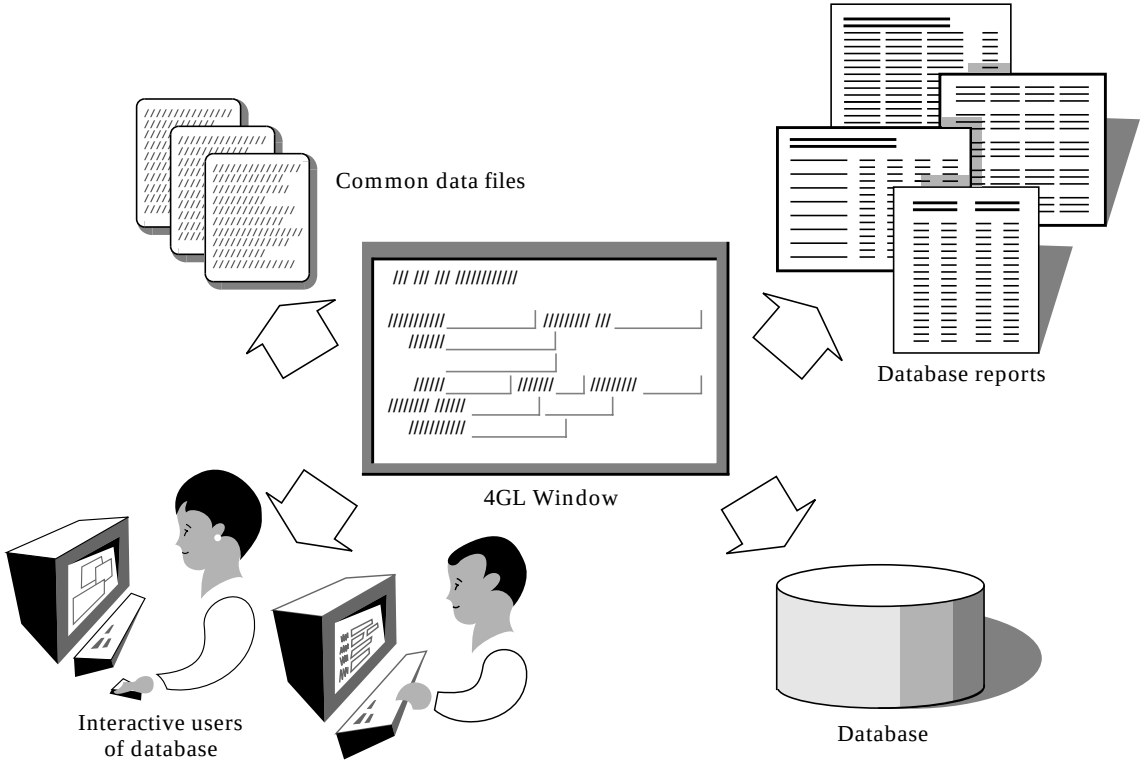
2

Interfaces of INFORMIX-4GL

Overview

Multi-user application programs you write using **INFORMIX-4GL** have four primary interfaces to other software:

- Accessing a database through a database engine.
- Communicating with users through a terminal.
- Accessing sequential files through the host operating system.
- Generating reports that can then be sent to several destinations.



Database Access

Your **4GL** programs can access SQL databases using the appropriate **INFORMIX-OnLine Dynamic Server** and INFORMIX-SE database engines.

The database engine can be located on the same computer as the application program, but this may not be the case.

Once network connections are properly established, database access is transparent; you need not write any special code to make it happen. In fact, the same **4GL** program can work with a local database engine in the same computer on one occasion, and over a network to an engine in another computer on another occasion.

Access to Sequential Files

Your **4GL** application can use standard sequential, or flat, data files in the following ways:

- The UNLOAD statement writes selected rows of data to a specified file
- The LOAD statement reads a specified file and inserts its lines as rows in a database table
- The START REPORT or the REPORT statements can send output from a report to a specified sequential file or an operating system pipe
- The PRINT FILE statement can incorporate the contents of a specified sequential file into the output of a **4GL** report
- The DISPLAY statement can be used to write lines to the Application window. Using the host operating system you can direct these lines to a file or another program

Report Output

Your **4GL** program can generate powerful and flexible reports. The output of a report is a series of print lines. This output can be directed to any of several destinations:

- A screen
- A printer
- A host system sequential file, specified by its pathname
- A UNIX operating system “pipe” to another process.

The logic for generating reports is the same in all cases. The destination can be coded into the program or selected at execution time.

Report programming is covered in more detail in Part II, with examples in Part III of this book.

User Access

To provide for portability of code across different platforms, your program interacts with the user through windows with a fixed number of rows and character-width columns.

The user of your application may be using any of the following:

- A terminal-emulation program in a personal computer or workstation that is connected to a UNIX network.
- A character-based terminal connected to a UNIX system.

The key point is that no matter what the display device, the **4GL** user interface is programmed in terms of fixed-size characters; that is, so many columns across by so many rows down.

Forms are also portable across applications and platforms. You design screen forms through which your program can display data; then you compile the forms separately from the other parts of the program. Thus the compiled forms can be used with different programs.

You display a form with one program statement and fill it with data in another; you can prompt for user input from any or all fields with a third. You can easily open additional display areas for alerts and other information. Interaction through a hierarchical set of ring menus (horizontal menus) is also fully supported.

Using Forms and Menus

The typical **4GL** program is designed around a hierarchy of screen forms and menus. The program presents a main menu containing a list of major activities, for example query, insert, or update. The user makes a selection of one activity and the program displays the form for that activity. When the activity ends, the program redisplays the main menu.

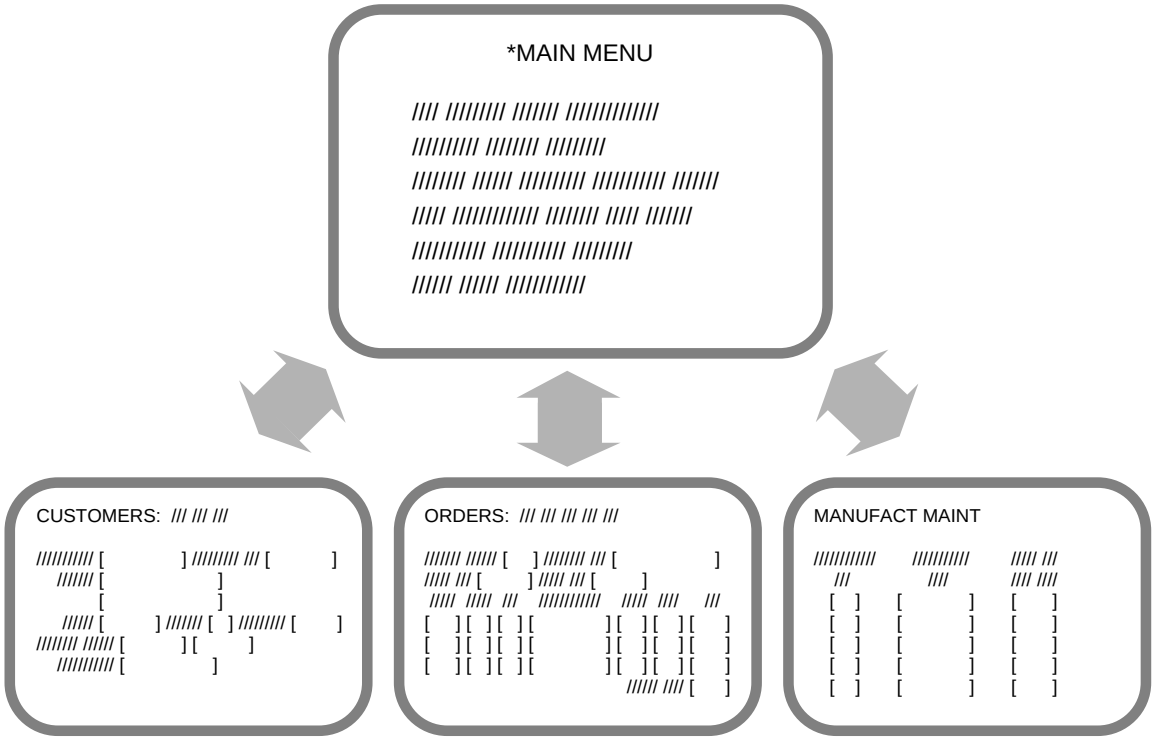


Figure 2-1 High-level view of a program with three major forms

The key characteristic of a **4GL** program is that the program chooses which screen elements the user interacts with at any particular time. This predictability leads to an easily maintained linear program structure and simple program logic.

Summary

You use **4GL** to write programs that connect key elements in the following informational structure:

- The database
- The interactive users of the database
- Common data files
- Database reports

In dealing with the database, you use the industry-standard Structured Query Language, that you can augment with custom programming logic.

As you will see in subsequent chapters, your programs have a simple method for access to common sequential files, and a nonprocedural, task-oriented method of getting information from a database and defining and producing reports.

You can use **4GL** to program elaborate user interactions based on forms and menus. This user interface is character-oriented; that is, output is displayed in a fixed number of evenly spaced lines, each of which contains a fixed number of monospace characters. This approach allows applications written in **4GL** to run without modification on supported platforms.

Chapter

3

The INFORMIX-4GL Language

Overview

This chapter provides a brief tour of **INFORMIX-4GL** as a language. The purpose is to give you an idea of what **4GL** code looks like. There are many short examples in the following chapters and several complete examples supplied on-line with the product. (For additional details on the **4GL** programming language be sure to see Chapter 8, "Using the Language.")

As a programming language, **4GL** has several important features:

- It is a procedural language, with facilities for structured programming.
- It is a nonprocedural ("fourth-generation") language with regard to:
  - Database access
  - Reports
  - Form-based user interaction
- It is C-like in design, but much easier to read, write, and support.

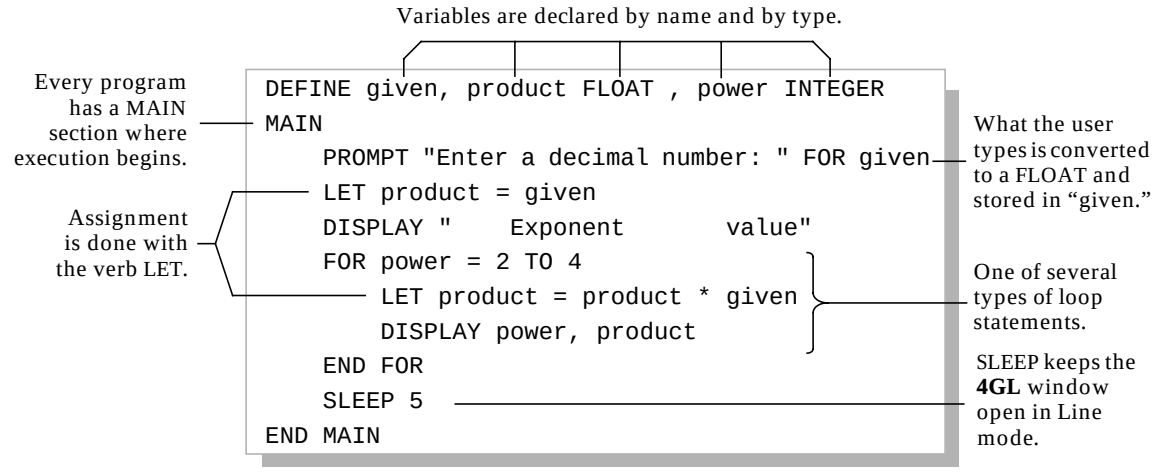
A Structured, Procedural Language

4GL is a general-purpose programming language for creating structured programs, the way you might use Pascal, C, COBOL, or PL/1. Like these languages, **4GL** offers statements you use to perform the following tasks:

- Declare variables of different types
- Calculate values and assign them into variables
- Declare functions
- Apply functions to data
- Display the contents of variables on the screen

Also like other languages, **4GL** has control statements you use to define choices, loops, and statement blocks of code.

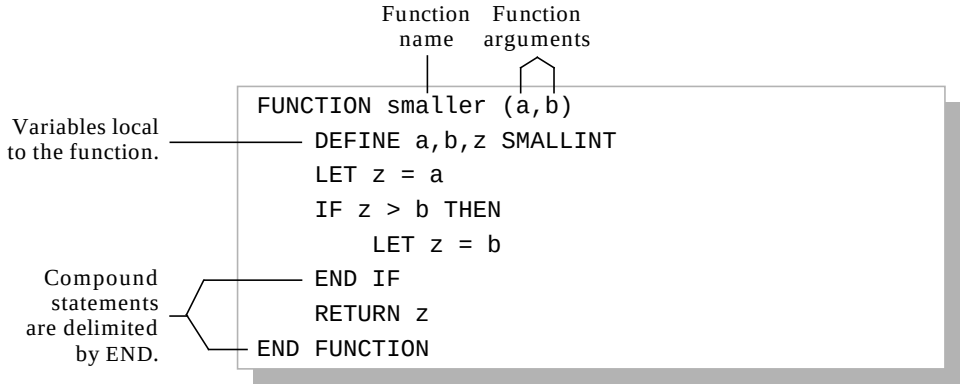
Here is a short, complete program in **4GL**. It prompts the user for a number and displays the square, cube, and fourth power of the number.




```
DEFINE given, product FLOAT , power INTEGER
MAIN
    PROMPT "Enter a decimal number: " FOR given
    LET product = given
    DISPLAY "    Exponent       value"
    FOR power = 2 TO 4
        LET product = product * given
        DISPLAY power, product
    END FOR
    SLEEP 5
END MAIN
```

4GL is *not* sensitive to the letter case of a source statement. The use of all-capitals for language keywords such as DEFINE and MAIN is merely a convention used in these manuals. You are free to write keywords in lowercase, or any combination of capitals and lowercase you prefer.

The **4GL** language supports structured programming. Its design encourages you to build your program as a family of simple, reusable functions with well-defined interfaces. The following function returns the lesser of two integers.




```
FUNCTION smaller (a,b)
    DEFINE a,b,z SMALLINT
    LET z = a
    IF z > b THEN
        LET z = b
    END IF
    RETURN z
END FUNCTION
```

As described in the next chapter, **4GL** also has features making it easy to assemble large programs from many small source modules.

A Nonprocedural, Fourth-Generation Language

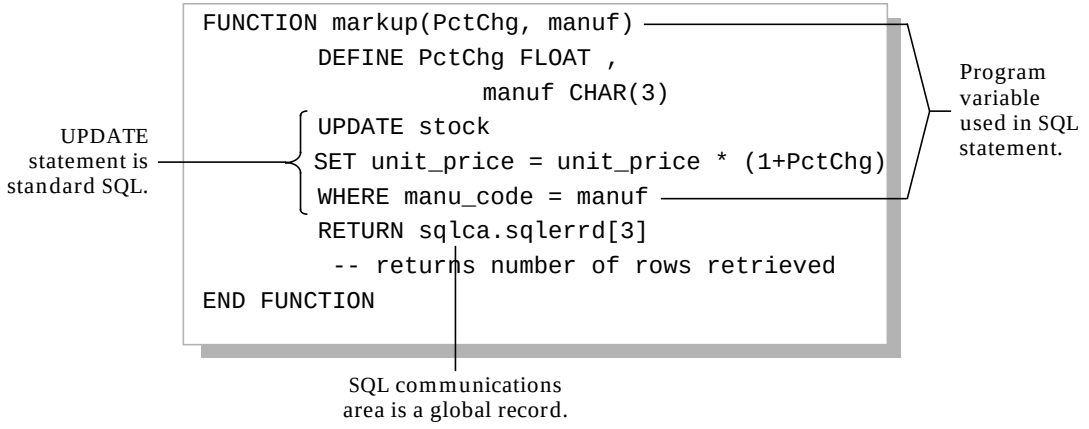
4GL is a nonprocedural, or "fourth generation" language in three important areas:

- Database access
- Report generation
- User interaction

In these areas you specify what is to be done, while **4GL** (or the database engine) takes care of the actual sequence of events.

Database Access

4GL includes all Informix 4.1-level Structured Query Language (SQL) statements as native statements of the language. The following function applies a change in price to all the items in the **stock** table that are purchased from a particular manufacturer. The function returns the number of rows changed to the calling routine.



It is the nature of many SQL statements to be *nonprocedural*; that is, you use the statements to specify what is to be done, and the database engine determines how to do it.

However, using **4GL** you can write code that applies sophisticated logic to the results of the SQL statements. For example, the function in the preceding example could contain statements to validate the argument before applying the change, or to verify the authorization of the current user to perform such a change.

Report Generation

A report is an organized display of data that has been extracted from the database. Reports are often tabular in form, can be voluminous in size, and designed for printing with page headings and footings. Frequently reports are produced by noninteractive, "batch" programs, perhaps run at night. Any **4GL** program can produce a report.

4GL divides the task of producing a report into two parts. One part, which may require procedural logic, is the production of the rows of data that go into the report. The second part is the logic within the report itself: your decisions as to how to format header and footer lines, detail lines, control breaks, and display subtotals.

You write the logic of the report in a nonprocedural form, as a collection of code blocks that are called automatically as needed. For example, your code block for a group subtotal can be executed automatically on each change in the value of the group control variable. Your code block for a page footer is called automatically at the bottom of each page.

Thus, as you design and code the logic of a report, you think about each part of the report in isolation. **4GL** supplies the "glue" logic that invokes the report sections as required.

Examples of reports are shown in subsequent chapters, particularly in Chapter 10, "Creating Reports." One key point: the part of the program that produces the data rows does not need to arrange the rows. **4GL** can automatically collect generated rows and sort them before presenting them to the report code, if that is necessary.

User Interaction

4GL contains extensive support for writing interactive programs. The following illustration shows a typical screen form with a menu, together with the *pseudocode* that would be used to display it. (Example 19 in *INFORMIX-4GL by Example* contains the complete source code of this program.)

```
View Customers:   Query  First  Next  Last  Exit
Display next customer in selected set
-------------------------------------------Press CTRL-W for Help----------


  Customer Number: [        101] Company Name: [All Sports Supplies ]
   Address: [213 Erstwild Court  ]
            [                    ]
      City: [Sunnyvale      ]   State: [CA]  Zip Code: [94086          ]

   Contact Name: [Ludwig         ][Pauli          ]
      Telephone: [408-789-8075      ]
```

Gets precompiled form from disk

Displays form fields and labels on screen

Displays menu choices and specifies the code to execute when each choice is selected

```
OPEN FORM f_cust FROM "f_cust"
DISPLAY FORM f_cust
MENU "New Customers"
    COMMAND "Query"
        CALL queryCust()
    COMMAND "First"
        CALL firstCust()
    COMMAND "Next"
        CALL nextCust()
    COMMAND "Last"
        CALL lastCust()
    COMMAND "Exit" KEY(Escape,CONTROL-E)
        EXIT MENU
END MENU
```

You describe a screen form in its own source file and compile it separately from program code. Since forms are independent of **4GL** programs, they are easy to use with many different **4GL** applications.

You can fill some or all of the fields of a form from program variables with a single statement. With another statement you can open up form fields for user input with the entered data returned to program variables. For detailed validation, you can attach blocks of code to form fields. The code is executed when the cursor leaves or enters a field.

You can open a **4GL** *window*, optionally containing another form, over the top of the current **4GL** window, and then restore the original display. There is support for scrolling lists of data both for display and editing. These features are covered in subsequent chapters.

Summary

4GL has all the features of a standard, structured programming language. It goes beyond such languages as Pascal or C in that it supports nonprocedural, task-oriented ways of programming database access, report generation, and user interaction.

Chapter

4

Parts of an Application

Overview

You typically use **INFORMIX-4GL** to build an *interactive database application*, a program that mediates between a user and a database. The database schema that organizes data into relational tables gives shape to one side of the program. The needs of your user shape the other side. You write the program logic that bridges the gap between them.

Such a program has many parts that you prepare with the help of the **4GL** Programmer's Environment. The main parts of an application are:

Form source files	You specify the user interface to your application using editable files that specify the appearance of a form on the screen and the data types and attributes of the fields in the form.
Form object files	Your form specifications are compiled into a binary form by using FORM4GL, the **4GL** form compiler. These are loaded by the program during execution.
Message source files	You write the text of help messages and other messages separately from your programs. You can use a common set of messages for multiple programs, and you can change these (for instance to support a different language) independently of the programs.
Message object files	Your messages are indexed for rapid access by **mkmessage**, the **4GL** message compiler. Like a form, a compiled message file can be used by many different programs.
Program source files	You write your program as one or more files of **4GL** source code.
Program object files	Your sources are compiled into C code or p-code executable files, depending on the version of **4GL** you are using. (For a brief description of the two versions, see "Two Versions of 4GL" on page 1-5.)

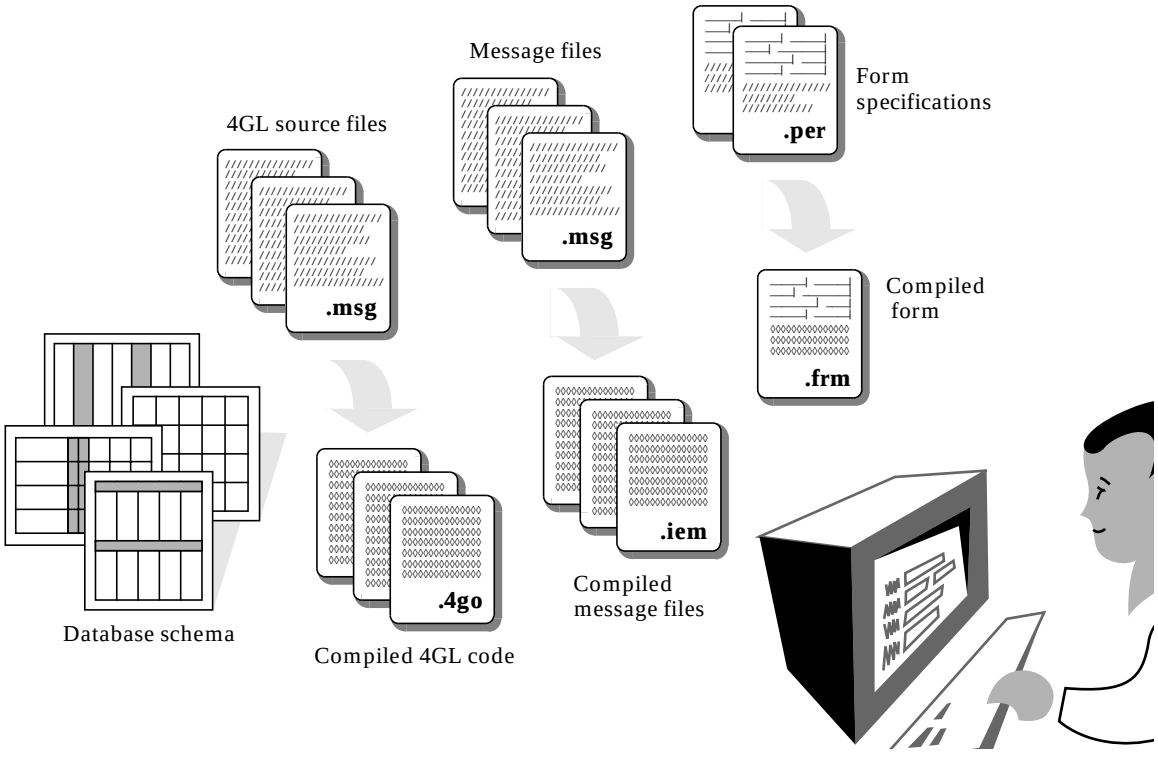


Figure 4-1 Parts of a 4GL application

The Database Schema

In the structure of the database, either you or another person acting as the Database Administrator (DBA), carefully design a representation of the real world, or an important part of it.

This picture of the real world is expressed in the form of *tables* of information, each containing categories of information, called *columns*. It is not simple to make the proper choice of columns and to group them into tables so that the data can be used efficiently and reliably as your needs change. In fact, many books have been written on the subject of how best to design a database structure, usually referred to as a *schema*.

The database schema affects your **4GL** program in many ways:

- You can have program variables that have the same data types and names as database columns. **4GL** makes this easy by letting you declare a variable as being LIKE a column in a table. When the program is compiled, the compiler queries the database for the appropriate data type.
- Your program can contain SQL statements that refer to names of tables and columns. Any change in the schema may require you to examine these statements and possibly change them.
- The logic of your program may depend on the schema. For example, if your program has to change the schema of several tables in performing a certain operation, you may want to use explicit database transactions; while if the data is arranged so that only a single table needs changing, you can avoid this.

In general, before you start work on a large application, you should make sure that the database schema is workable and that you and others who will be using the database understand it well.

Form Specifications and Form Files

In many applications the user interface is defined by *forms*. A form is a fixed, functionally organized arrangement of *fields* (areas where you can display program data and the user can enter text) and *labels* (static, descriptive text).

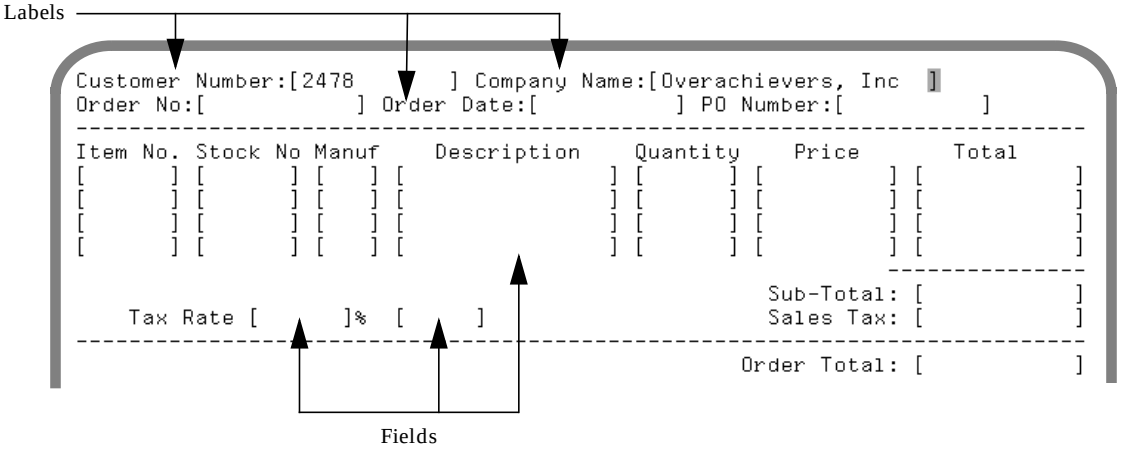


Figure 4-2 A form containing labels, fields, additional text, and a screen array

The user of a program does not know about the database schema nor your carefully-designed program logic. As the user sees it, the forms and the menus that invoke them *are* the application. The arrangement of fields and labels, and the behavior of the forms as the user presses different keys and selects different menu options, create the personality of the program.

4GL form specification files are ASCII files. You can use an ASCII text editor to design the labels and fields for your form. Here is a portion of the form specification file used to create the preceding form:

```
SCREEN
{

Customer Number:[f000       ] Company Name:[f001                ]
Order No:[f002       ] Order Date:[f003      ] PO Number:[f004      ]
-------------------------------------------------------------------------------
Item No. Stock No Manuf    Description    Quantity    Price       Total
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
                                                               ---------------
                                                 Sub-Total: [f012       ]
    Tax Rate [f013  ]%  [f014 ]                  Sales Tax: [f015       ]
-------------------------------------------------------------------------------
                                               Order Total: [f016       ]
}
TABLES
customer orders items stock state
ATTRIBUTES
f000 = orders.customer_num;
f001 = customer.company;
...
```

To see the entire text of this example, see Example 11 in *INFORMIX-4GL by Example*.

After specifying a form, you compile it with FORM4GL, the **4GL** form compiler. The result is a portable binary file that can be opened and displayed from any **4GL** program on any platform supported by **4GL**.

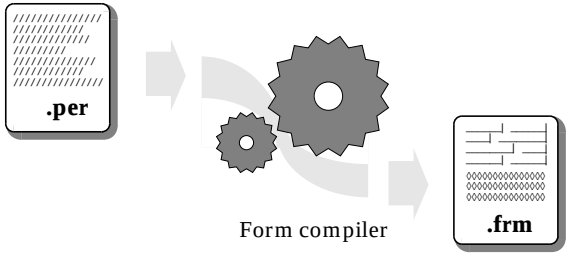


Figure 4-3 The form compilation process

Compiled forms are independent of the programs that use them, so you can use the same forms in different applications for a consistent “look and feel.”

Since forms are so important to the users of your application, you should consider designing the main forms before any other part of the program. You can quickly prototype programs that display forms so your users can give you their opinions.

Here is the text of the program that displayed the form on page 4-6:

```
MAIN
    OPEN FORM fx FROM "f_orders"
    DISPLAY FORM fx
    DISPLAY "2478" TO orders.customer_num
    DISPLAY "Overachievers, Inc" TO customer.company
    SLEEP 60
END MAIN
```

Form Design

Although user interface design is both an art and a science, most form designers are neither scientists nor artists. Here are a few points to consider when creating forms for your application:

- Is the purpose of the form clear from its title and the title of the menu command that invokes it?
- Are fields in the form arranged in the same logical order that the user typically follows in transcribing or describing the information?
- Is the same information always given the same label in every form in which it appears?
- Are form labels consistent in style and content?
- Is the relationship between various forms as clear as possible?
- Is it obvious how to complete the form and what fields are required?

Field Entry Order

With **4GL** you can constrain the user to entering fields in a preset order, or you can permit entry into fields in any order desired by the user. Since application and form requirements differ, you can control these factors on a form-by-form basis.

Program Source Files

You express the logic of your application with **4GL** statements in program source files.

Organization of a Program

If you are using the **C Compiler Versio**n of **4GL**, the files containing executable **4GL** statements require the file suffix **.4gl** (otherwise the program compiler cannot find them). However, if you are using the **RDS** version of **4GL**, you can omit the **.4gl** file suffix.

Executable statements are organized as *program blocks*. A function is a unit of executable code that can be called by name. In a small program, you can write all the functions used in the program in a single file. As programs grow larger, you will usually want to group related functions into separate files, or *modules*, with the declarations they use.

Each source file usually reflects a self-contained unit of program logic; source files are sometimes called *source modules*.

Execution of any program begins with a special, required *program block* named MAIN. The source module that contains MAIN is called the main module. Here is a small but complete **4GL** program:

```
MAIN
    CALL sayIt()
END MAIN

FUNCTION sayIt()
    DISPLAY "Hello, world!"
END FUNCTION
```

This single module contains the MAIN program block, delimited by the statements MAIN and END MAIN, and one other function named **sayIt**( ).

A single function cannot be split across source modules. However, since the program above has two functions it could be split into two source modules. The first would look like:

The MAIN program block

```
MAIN
    CALL sayIt()
END MAIN
```

The second module could contain the three lines of function **sayIt**( ) just as shown above. It could also contain data or other functions related to **sayIt**( ), if there were any.

Functions are available on a global basis. In other words, you can reference any function in any source module of your program.

The Globals File

In **4GL** programs, global variables (variables available to more than one source module) are declared in a *globals* file and referenced through the GLOBALS statement by each **4GL** module that uses them. For more information on local and global variables, see "Variables and Data Structures" on page 8-8.

Program Object Files

The **C Compiler Version** and the **Rapid Development System** of **4GL** each provide its own source code compiler. These compilers generate distinct, executable forms of the same program:

- **c4gl**, the *C code compiler* provided with the **C Compiler Version**, generates code that can be executed directly by the hardware of the computer after an executable program is created.
- **fglpc**, the *p-code compiler* provided with the **Rapid Development System**, generates hardware independent code—code not directly executable by the operating system or GUI—that is interpreted by the **4GL** *p-code runner*.

Both can take the same **4GL** source code as input. If the C code option is selected, the output is a C language object file. When this file is linked with other **4GL** libraries and perhaps C object modules, an independent executable program is produced.

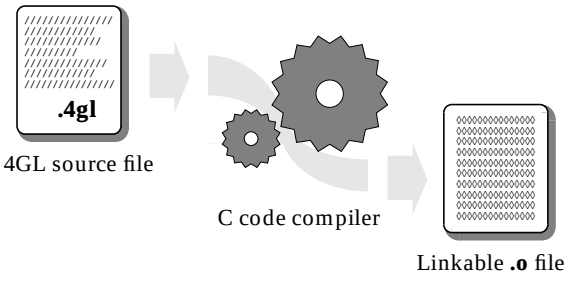


Figure 4-4 The C object code generation process

If the p-code option is chosen, p-code intermediate object files are created that are executable under the **4GL** runner. P-code stands for *pseudo-machine code*.

Both types of compiled files are binary. That is, the file is not printable or editable as text. All of the modules of an application have to be compiled to the same form; that is, the executable version of your program cannot mix C code and p-code units, although the p-code runner can be customized to call C object modules.

For detailed coverage of the steps required to compile **4GL** source files of all types, as well as using C with **4GL**, see Chapter 1 in *INFORMIX-4GL Reference*.

P-code Object Files

You can use **fglpc**, the p-code compiler, to translate a source module into p-code. The output of the p-code compiler will have the file suffix **.4go**.

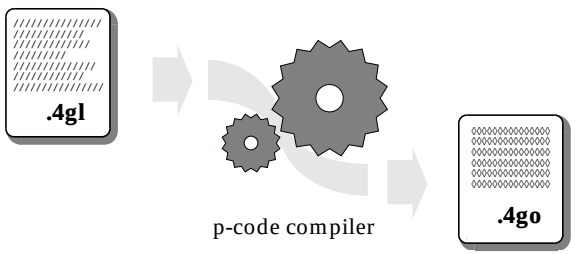


Figure 4-5 The p-code object code generation process

When your application is in several source modules, you first compile each separate module to p-code. Then you can concatenate the individual p-code files using a utility (**cat** in UNIX environments) to make the executable **.4gi** program file.

Note: *When you use the Programmer's Environment to build and maintain your program, module linking is done automatically for you.*

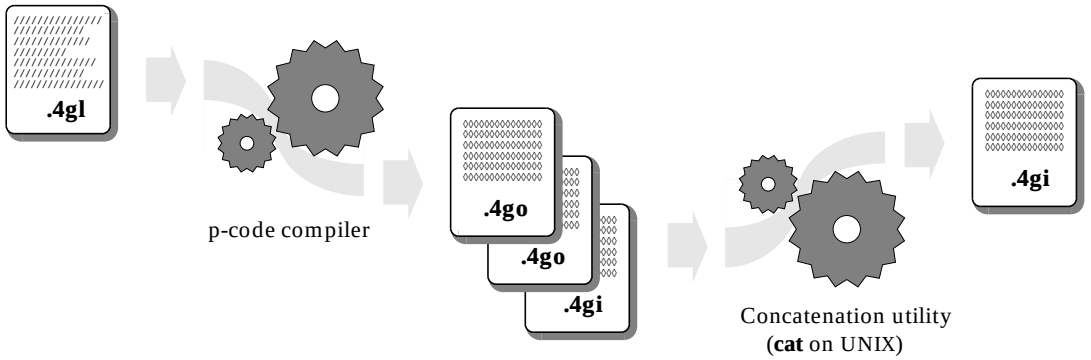


Figure 4-6 Steps to creating an executable program under the p-code runner

To execute a **.4gi** file you call the **4GL** *p-code runner*. Use **fglgo** to execute the p-code instructions in the file, bringing your program to life, metaphorically speaking.

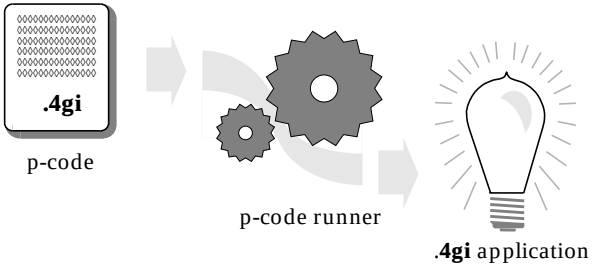


Figure 4-7 Components of a runnable p-code application program1

C Code Object Files

You can use the Informix C code compiler, **c4gl**, to translate a source module directly into machine code. This is done in three primary stages:

1. The module is translated to **INFORMIX-ESQL/C** source code.
2. The **ESQL/C** processor converts that to C source.
3. The compiler translates to C object code for your computer.

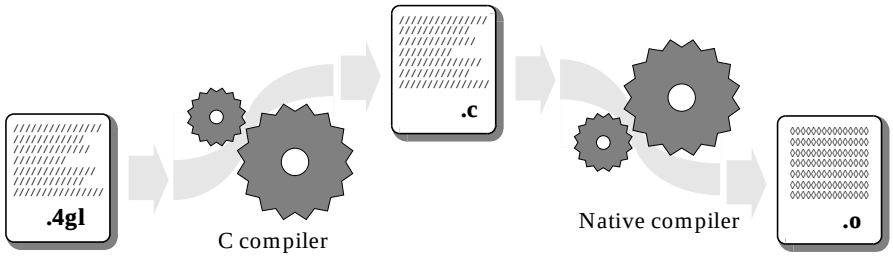


Figure 4-8 ***Steps to creating a C object file***

From the operating system command line a single call to the **c4gl** command performs all the steps, or all these steps can be automatically accomplished through the Progammer's Environment.

On UNIX systems the default extension is **.4ge**. However, it is not required. You may name your executable applications anything you like.

The C file from a single source module is compiled to an **.o** file. Several **.o** files for a multi-module application can be combined into a single executable file through the Programmer's Environment or using another concatenation utility. In fact the **c4gl** command line accepts any combination of **.4gl** files, **.ec** files, **.c** files, and **.o** files to produce a single **.4ge** executable file.

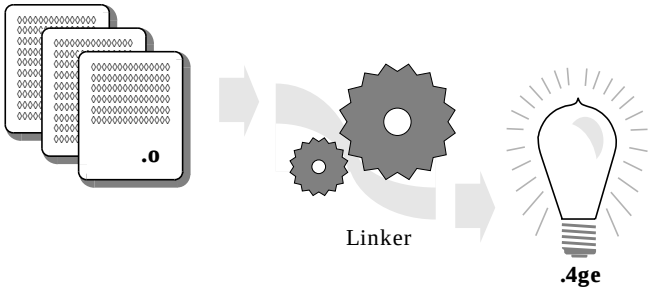


Figure 4-9 ***Steps to creating an independently executable 4GL program***

Example Programs

Now that you know the parts of a **4GL** program, you should look at a few of them to see what they are like. A number of example programs are distributed with **4GL**. And, the companion volume *INFORMIX-4GL by Example* contains 30 complete and annotated **4GL** programs.

Chapter

5

The Procedural Language

Overview

INFORMIX-4GL is a fourth-generation programming language. However, it uses some concepts based on procedural languages (such as Pascal and C).

This chapter describes the concepts based on procedural programming; in particular, it describes how to:

- Declare variables
- Organize statements for decisions and looping
- Handle exceptions

Declaration of Variables

4GL has a flexible mechanism for declaring program variables. You specify a type for every variable, but there is automatic conversion between many data types. Data can be organized as records or arrays. It can be allocated statically or dynamically and can be declared at several scope levels.

Data Typing

4GL is a "strongly typed" language. That is, whenever you declare a program variable you must specify its data type, for example, INTEGER or CHAR. The compiler ensures that only data of that type can be stored in that variable. Here is an example declaration of several types of variables:

```
DEFINE
    aFloat FLOAT
    oneInt, anotherInt INTEGER,
    aString CHAR(20)
```

The data types supported by **4GL** for program variables include all the data types that are allowed in a column in a database except SERIAL. This includes BYTE and TEXT, the Binary Large Object (blob) data types.

An important point to note is that **4GL** defines a specific NULL value for every data type. Null means "unknown," rather than 0, which is a precise value. You can assign NULL to any variable, and you can test any variable for NULL content. This is necessary to properly support database access, since NULL is a distinct value (or, to be precise, *non-value*) in a database table.

Automatic Data Conversion

With some strongly-typed languages, it is an error to assign a value of one type into a variable of a different type. By contrast, **4GL** allows you to assign any value into any variable provided that there is a reasonable way of converting between the type of the value and the type of the variable.

```
LET aFLoat = 2.71828
LET oneInt = aFloat     -- assigns integer 2
```

The LET statement is used for assignment. The first of the preceding LET statements assigns a literal decimal number into a variable of type FLOAT. (In the above code fragment, the names **aFloat** and **oneInt** were declared in the example on page 5-3.)

In the second LET statement a FLOAT value—the contents of the variable **aFloat**—is assigned to integer variable **oneInt**. **4GL** converts the FLOAT value into an integer to match the type of the receiving variable. To do so, the fractional part of the floating point number is truncated.

One conversion that is often used is between a character value and almost any other type.

```
LET aString = aFloat-- assigns "2.71828" to aString
```

The above statement assigns a FLOAT value to a variable whose type is CHAR (character string). **4GL** converts the numeric value to characters and assigns that string value to **aString**.

4GL will also attempt to convert a character string back into a numeric or other type. The following statement assigns a string of numeric characters to a FLOAT variable:

```
LET aString = "3.141592"
LET aFloat = aString    -- assigns 3.141592 into aFloat
```

If the characters in the string can be interpreted as a literal value of the type needed, the conversion is done. Most data types have a printable, character representation, and **4GL** converts automatically between the printable form and the internal form.

Of course there are some cases when conversion is not allowed; for example, the BYTE and TEXT data types cannot be converted into any other type. Such errors are detected at compile time.

Some conversions can only be found to be impossible at execution time. The following example fails in its attempt to convert a large floating-point number to integer.

```
LET aFloat = 2E12   -- about 100 times the maximum integer size
LET oneInt = aFloat -- this causes a runtime error
```

You can manage such errors by:

- Anticipating them and inserting programmed tests to avoid them
- Trapping the error at execution time
- Letting the run-time error terminate the program with an appropriate message

See Chapter 3 in the *INFORMIX-4GL Reference* manual for a table that identifies all the pairs of data types for which **4GL** performs automatic conversion.

Data Structures

Besides simple typed variables, you can organize program data into *records* and *arrays* (also known as *data structures*).

Records

A record is a group of variables that are treated as a unit. Each *member* of a record has a name. The following code fragments defines a record:

```
DEFINE stockRow, saveStockRow RECORD
            stock_num   INTEGER ,
            manu_code   CHAR(3) ,
            description CHAR(15) ,
            unit_price  MONEY(8,2) ,
            unit        CHAR(4) ,
            unit_descr  CHAR(15)
        END RECORD
```

This statement defines two program variables. Their names are **stockRow** and **saveStockRow**. Each variable is a record with six *members*.

The member named **manu_code** is a three-character string. You refer to this member of the **stockRow** record as **stockRow.manu_code**. The parallel member of the other record, **saveStockRow**, would be called **saveStockRow.manu_code**.

The members of these records are all simple data types. However, a record can contain members that are other records or arrays.

Another interesting aspect of the record is that it contains one member for each column in the **stock** table of the stores demonstration database. Because it is so common to define a record that matches one-for-one to the columns of a database table, **4GL** has an easier way of doing this.

```
DEFINE stockRow, saveStockRow RECORD LIKE stock.*
```

This statement causes the **4GL** compiler to refer to the database, extracts the names and types of all the columns, and inserts them in the program. In this way, you can ensure that the program will always match the database schema.

You can also fetch a row of a database table into such a record.

```
SELECT * INTO stockRow.* FROM stock
    WHERE stock_num = 309 and manu_name = "HRO"
```

You can assign the contents of all the members of one record to another record with a single statement.

```
LET saveStockRow.* = stockRow.*
```

You can do this even when the two records are not defined identically. The assignment is done member-by-member. As long as the records have the same number of members, and data from each member on the right can be converted to the type needed by the corresponding member on the left, you can assign the contents of one record to another.

Arrays

An array is an ordered set of elements all of the same data type. You can create one-, two-, or three-dimensional arrays. The elements of the array can be simple types or they can be records.

```
DEFINE stockTable ARRAY[200] OF RECORD LIKE stock.*
```

This array variable is named **stockTable**. It contains 200 elements each of which is a record with as many members as there are columns in the **stock** table in the database. One of those columns is named **stock_num**. You would access the **stock_num** member of the 52nd element of the array by writing **stockTable[52].stock_num**.

The first element of any array is indexed with subscript **1**. (This differs from the C language, where the first element is always number zero.) The subscript value that selects an element can be given as an expression. Expressions are described in the section titled "Expressions and Values" on page 8-18.

Data Allocation

4GL supports the allocation of program variables either statically, as part of the executable program file, or dynamically, at execution time. You choose the method to use by the location of your DEFINE statement in the source module.

```
DEFINE greeting CHAR(5)
DEFINE audience CHAR(5)

MAIN
    LET greeting = "Hello"
    LET audience = "world"
    CALL sayIt()
END MAIN

FUNCTION sayIt()
    DEFINE message CHAR(40)
    LET message = greeting , " " , audience, "!"
    DISPLAY message
END FUNCTION
```

Module-level variables are allocated statically.

Local variables are allocated dynamically when the function is entered.

The topics of data allocation, scope of reference, program blocks, and functions are considered in detail in *INFORMIX-4GL Reference*. In summary:

- Variables that you declare outside of any function are at the *module level* of the program. They are allocated statically, at compile time, and become part of the *program image*. They may be referenced by any function in the source file that follows the definition.
- Variables that you declare within a function are *local* to the function. New copies of these variables are created each time the function is called. They are discarded when the function exits.

Scope of Reference

The *scope* of a variable name is the range of program statements over which it can be used. **4GL** has several scope levels. In summary, the levels are:

local Names declared within a program block are local to the program block. Their scope is from the point of declaration to the end of the program block. The variables are created each time the function, report, or MAIN statement is entered and cease to exist when the program block terminates.

module Names declared outside of any program block are local to the module. Their scope is from the point of declaration to the end of the module.

global Variable names declared with the GLOBALS statement are global to the program. The GLOBALS statement is also used to specify a file of globally available variable declarations in each source module that uses the global variables.

Decisions and Loops

4GL has statements for looping and decision-making comparable to other computer languages. These statements are covered in "Decisions and Loops" on page 8-26, and in the *INFORMIX-4GL Reference*. Here is a brief summary:

IF...THEN...ELSE	Tests Boolean (yes / no) conditions
CASE	Make multiple-choice decisions
WHILE	Used for general loops controlled by a Boolean condition
FOR	Used for loops that iterate over an array

There is also FOREACH, a special loop used for database access, covered under "Row-by-Row SQL" on page 9-5. These control statements can be nested one within the other to any depth. A key point is that their syntax is very simple and regular:

- Every statement that can contain other statements is closed with its own END statement (IF is closed with END IF, CASE with END CASE, and so on).
- Every looping statement has a specific optional early exit statement (you leave a WHILE with an EXIT WHILE, a FOR with an EXIT FOR, and so on).
- No special punctuation is needed in **4GL** code. You do not need to put semicolons between statements as in C or Pascal (although you may if you like). Nor do you need to write parentheses around a Boolean condition as in C (but you may if you like).

4GL also supports GOTO...LABEL statements.

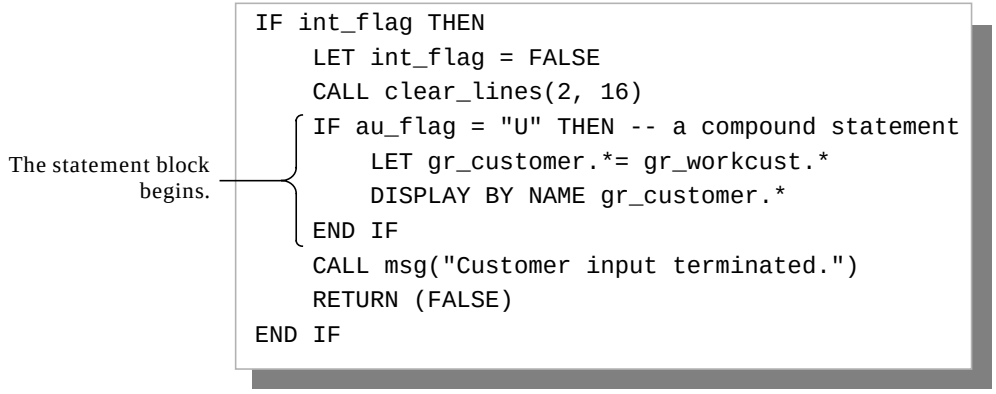
Statement Blocks

Many **4GL** statements such as LET and CALL are *atomic*; that is, they contain only themselves. Others are compound; that is, they can contain other statements. The most common examples of compound statements are:

IF-THEN-ELSE	THEN and ELSE each introduce a block of statements. The ELSE block is optional.
FOR	The body of a FOR loop is a block of statements.
WHILE	The body of a WHILE loop is a block of statements.
CASE	WHEN and OTHERWISE each introduce a block of statements.

Statement blocks can be nested; that is, one compound statement can contain another.

Here is a code fragment from Example 9, *INFORMIX-4GL by Example*, that contains a nested IF statement block:



Comments

You can comment your **4GL** code using double hyphens (--) or the pound sign (#) for individual lines (see the previous code fragment) or curly braces ({ and }) for multiple contiguous lines of code.

Exceptions

An *exception*—also known as an *error condition*—is an unplanned event that interferes with normal execution of a program. Usually an exception is not expected to occur in the normal course of events, and it requires special action when it does occur.

Kinds of Exceptions

There are several kinds of exceptions; they can be categorized this way:

run-time errors Errors in program statements detected by **4GL** at run time. These errors are divided into groups based on the kind of statement:

- *File I/O* Errors using files managed by the host operating system.
- *Display* Errors using the screen.
- *Expression* Errors in evaluating **4GL** expressions; for instance, a data conversion error or an invalid array subscript.
- *SQL error* Errors reported by the database engine.

SQL end of data Warnings that you have reached the end of a set of rows being fetched through a cursor.

SQL warning A warning condition reported by the database engine.

external signals Events detected by the host operating system. An external signal is usually not directly related to the program's statements. Two common external signals that **4GL** can handle are Interrupt (CONTROL-C) and Quit (CONTROL-\).

Why Exceptions Must Be Handled

Exceptions are *unplanned* in the sense that they are events of low probability. Usually your program cannot predict or control when they will occur. This does not mean that they are always *unexpected*. For example:

- You are certain that there will be an end to any selection of rows, it is just that you do not always know which row will be the last.
- You are sure that some user will eventually try to cancel an operation, but you do not know when.

Alas, even run-time exceptions must always be anticipated, no matter how carefully you write your code.

Because program exceptions are sure to happen, you must design the program to handle them in a rational manner. But because they are of low probability, you want to handle them:

- Away from the main line of processing, so the code for the normal, expected sequence of events is clear and readable.
- With a minimum of overhead at execution time.

How Exceptions Are Handled

Unless you establish some way of handling exceptions, **4GL** takes the following actions when one occurs:

run-time error	the program is terminated
run-time warning	execution continues

When a run-time error is encountered, any SQL transaction in progress is automatically rolled back. Then **4GL** writes a message to the error log, which is by default the screen. (You can establish an error log file on disk if you wish.)

Note: *If your database is ANSI-compliant, then by default the program tries to continue if a run-time error is encountered.*

For many exceptions, the default response is correct; the program should be terminated with a message. But there are other types of exceptions that you can anticipate and handle with **4GL** statements including:

DEFER	establishes your program's policy for handling external signals that **4GL** recognizes. The DEFER statement can be used to instruct your executing program to set built-in global variables—rather than terminate the program—when an Interrupt or Quit signal is generated by the user. By polling the variable you can determine when the user is ready to end a routine and respond accordingly and in an orderly manner to the request.
WHENEVER ERROR	establishes your program's policy for handling an execution error.
WHENEVER ANY ERROR	tests for a response to an expression error.
WHENEVER WARNING	tests for SQL warning flags.

WHENEVER NOT FOUND can be used to determine when an executing SQL statement has encountered an "end of data" condition.

For more information on exception handling, see *INFORMIX-4GL Reference*.

Chapter

6

Database Access and Reports

Overview

One of the main reasons to use **INFORMIX-4GL** is the ease with which you can access and modify data in a database. SQL, the Structured Query Language that is the international standard for relational database access, is an integral part of the **4GL** language. Another reason to use **4GL** is the ease by which you can design and generate reports.

Using SQL in a 4GL Program

There are three ways you can use SQL in a **4GL** program:

1. Incorporate nonprocedural SQL statements as **4GL** program statements.

   Any SQL statement that does not return data to the program (for example, ALTER, CREATE, UPDATE, or REVOKE among many), can be written into the program for execution in sequence with other **4GL** statements. In many cases, you can use data from program variables as part of the statement.

   In addition, any SELECT statement that returns a single row of data can be written into the program and used to get data from the database and assign it to variables.

2. Use a *database cursor* to access a selected set of rows, one row at a time.

   A cursor contains a SELECT statement that may return multiple rows of data. You use an OPEN statement to start the selection. You use FETCH statements to fetch one selected row at a time, assigning the column data to variables. In this way, your program can scan a selection of rows, or bring all or part of the selection into memory and store it in an array.

   The FOREACH loop is another mechanism that can be used to open a cursor and fetch rows in sequence.

3. Use dynamic SQL to prepare new statements at execution time.

   You can assemble the text of an SQL statement from user input, then pass it to the database engine for execution. In this way, you can write an

application that adapts to the schema of the database or to user-selection criteria at execution time. SQL

The concepts behind all three of these methods are discussed in greater detail in Chapter 9 and Chapter 10. Additional discussion and examples are provided in the *Informix Guide to SQL: Tutorial*.

Note: *In this version of* **4GL** *you can include any level-4.1* SQL *statement. If you want to include* SQL *statements introduced after version 4.1, you must prepare the statement before including it in the program. (You prepare a statement by using the PREPARE statement.) For a complete list of supported* SQL *statements and information on preparing statements, see Chapter 3 in INFORMIX-4GL Reference.*

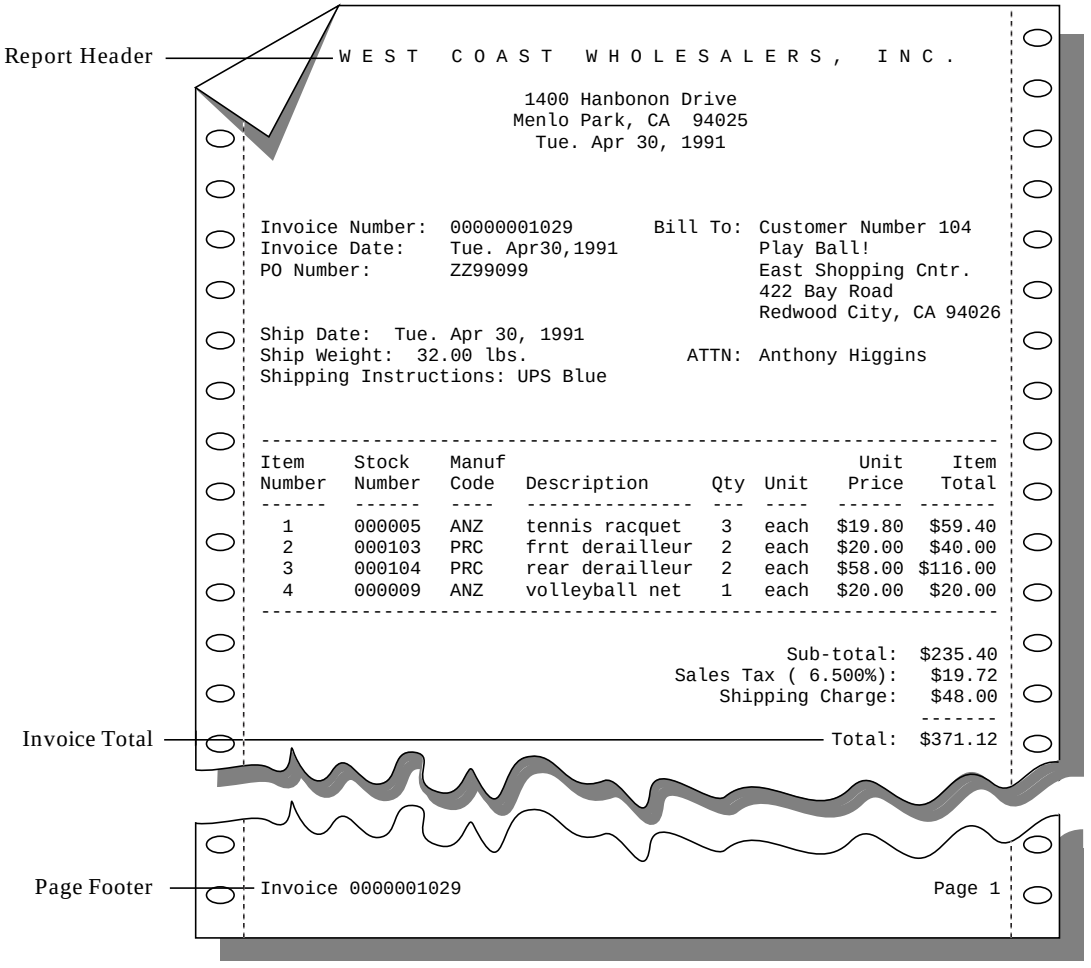
Creating 4GL Reports

A *report* is a display of data. A well-designed report is arranged so that the eye of the reader can easily pick out the important facts, such as column totals or sub-totals. The data can be shown on a screen or written to a printer or file.

A report is meant to be viewed on the screen or on paper, so it needs to be arranged in pages, often with a *heading* and a *footer* on each page, possibly with page totals.

The most important technique for making data clear to the eye is logical *layout*. The items in a report should almost always be arranged so that:

- The reader can quickly find an item.
- Related items are near each other.
- Groups of data with similar values fall together so that group totals and sub-totals can be calculated and shown.



It is much easier to make a report when the program logic that produces the report data is separate from the program logic that prepares the report itself. In **4GL**, this separation is a natural part of the language. Any program that produces a report can be cleanly separated into the part that produces the data, and a second part that formats the report.

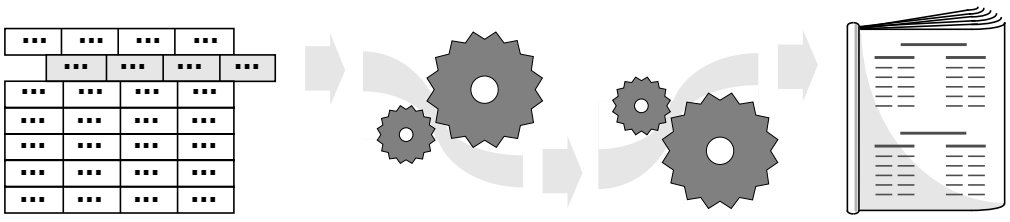


Figure 6-1 The report generation process

The Report Driver

The part of a program that generates the rows of report data (also known as input records) is called the *report driver*. It consists of the parts of a program that:

- Are concerned with the database and the user, not with the report format.
- Do not depend on the incidental features of a report, such as page length.
- May produce data for multiple reports simultaneously.
- May produce report data as a by-product of other operations, for instance a report on user productivity could be a by-product of carrying out user commands.

The key point is that the primary concern of the row-producing logic should be the selection of rows of data, not their arrangement or formatting.

In **4GL**, the actions of a report driver are as follows:

1. Use the START REPORT statement to initialize each report to be produced.
2. Whenever a row of report data is available, use OUTPUT TO REPORT to send it to the report formatter.
3. When the last row has been sent, use FINISH REPORT to end the report.

From the standpoint of the row-producing side, these are the only statements required to create a report. Multiple reports can be produced simultaneously. Although data for a report usually comes from a database, it can come from any source, including the user, calculations, or sequential files. Since the report driver pays no attention to issues of page formatting, grouping, or totalling, it produces report data as a by-product of other activities.

Your program is not required to produce rows in any special order. It is generally more efficient to produce them in their proper sorted order when possible (that is, by selecting database data in the desired order using the ORDER BY directive). However, you can produce them in any order, and leave the sorting to the report formatter.

The Report Formatter

You write the report formatter for each report in the same way as you write a **4GL** function. The code within a report program block consists of several sections:

DEFINE section	Here you define variables that are local to the report. A report can have its own local variables for subtotals, calculated results, and other uses.
OUTPUT section	Here you declare the size and margins of the report page. The OUTPUT section takes effect when the report is started.
ORDER section	Here you specify the required order for the data rows and whether or not the rows are provided to the report already ordered.
FORMAT section	Here you describe what is to be done at a particular stage of report generation.

The code blocks you write in the FORMAT section are the heart of the report program block and contain all its intelligence. You can write statement blocks to be executed for the following events:

- Top (header) of the first page of the report.
- Top (header) of every page after the first.
- Bottom (footer) of every page.
- Each new row as it arrives.
- The start of a group of rows (a *group* is one or more rows having equal values in a particular column). This statement block is often used for clearing totals and other accumulated values.
- The end of a group of rows. In this block, you typically print subtotals and other aggregate data for the group that is ending. You can call on aggregate functions like SUM and MAX for this information.
- After the last row has been processed.

You can use most **4GL** statements in the FORMAT section of a report; you cannot, however, include any SQL statements. For example, you can call other functions and interact with the user.

The key point is that **4GL** invokes the sections and blocks within a report program block non-procedurally, at the proper time, as determined by the report data. You do not have to write code to calculate when a new page should start, nor do you have to write comparisons to detect when a group of rows has started or ended. All you have to write is the statements that are appropriate to the situation, and **4GL** supplies the "glue" to make them work.

Chapter

7

The User Interface

Overview

Built into **INFORMIX-4GL** is a complete system of *character-oriented* user interaction. Being character-oriented, the same **4GL** applications can run on supported graphical user interfaces (GUIs), on personal computers, high-end workstations, and character-based terminals.

In all cases—character terminal, terminal emulation, and graphical—**4GL** lets you create highly flexible, portable, interactive, multi-user applications using screen forms and menus.

Note: *Version 6.0 of* **4GL** *is not available for the Windows or the Motif environment. If you have an earlier version of* **4GL** *that runs in one of these environments, see "Compatibility and Migration" in the introduction for information on using your existing forms and programs.*

Line-Mode Interaction

A **4GL** program can operate with its user interface in *Line mode* or *Formatted mode*. Line mode is the same typewriter-like mode of interaction that is used by UNIX shell scripts. You put the user interface in Line mode by executing a DISPLAY statement that does not specify a screen location. Here is a simple program that operates in Line mode.

```
MAIN
    DEFINE centDeg, fahrDeg DECIMAL(5,2)
    DEFINE keepOn CHAR(1)
    LET keepOn = "y"
    DISPLAY "Centigrade-to-fahrenheit conversion."
    WHILE keepOn == "y"
        PROMPT "Centigrade temp: " FOR centDeg
        LET fahrDeg = (9*centDeg)/5 + 32
        DISPLAY "old-fashioned equivalent: ", fahrDeg
        PROMPT "More of this (y/n) ? " FOR CHAR keepOn
    END WHILE
END MAIN
```

Because the first DISPLAY statement does not give a screen row and column for output, the screen is put in Line mode. Each line of display, and each prompt displayed by PROMPT, scrolls up the screen in the manner of a typewriter.

When you execute this program the interaction on the screen resembles the following, in which user data entry has been underscored.

```
    Centigrade-to-fahrenheit conversion.
    Centigrade temp: 16

    old-fashioned equivalent:  60.80
    More of this (y/n) ? y

    Centigrade temp: 28

    old-fashioned equivalent:  82.40
    More of this (y/n) ? n
```

You can use simple interactions of this kind for quick tests of algorithms. Line mode also has the virtue that you can redirect Line mode output to disk from the command line. The following program displays two columns from the **customer** table using Line mode output.

```
DATABASE stores2
MAIN
    DEFINE custno LIKE customer.customer_num,
            company LIKE customer.company
    DECLARE cust CURSOR FOR
                SELECT customer_num, company FROM customer
    FOREACH cust INTO custno, company
        DISPLAY custno, company -- Line mode display
    END FOREACH
END MAIN
```

You could execute this program from the command line, redirecting its output into a file, with a statement like the following (assuming the program has been compiled to **4GL** p-code in a file named **dump2col.4gi**):

```
fglgo dump2col | custcols.dat
```

The data could also be “piped” into another command. However, there is no equivalent input statement (PROMPT only accepts input from a real keyboard, not from a pipeline or redirected disk file).

Formatted Mode Interaction

Normally, **4GL** keeps the user interface in Formatted mode. That is, the output of the program is automatically formatted for screen display. The program's output is positioned by rows and columns. On character terminals, the initial **4GL** window is the same size as the screen and is referred to as the **4GL** *screen*.

Here is a brief summary of the **4GL** statements you use to manage the user interface in Formatted mode.

Statement	Purpose
DISPLAY…AT	Write data at specific rows and columns.
DISPLAY FORM	Display the background of a prepared form.
DISPLAY…TO	Write data into one or more fields of a form.
PROMPT	Prompt the user for a single value or a one-character response.
INPUT	Let user enter data into one or more fields or arrays of fields on a form.
CONSTRUCT	Let user enter search criteria into the fields of a form.
MESSAGE	Display a short message of warning or confirmation.
ERROR	Display a short message documenting a serious error to the screen.

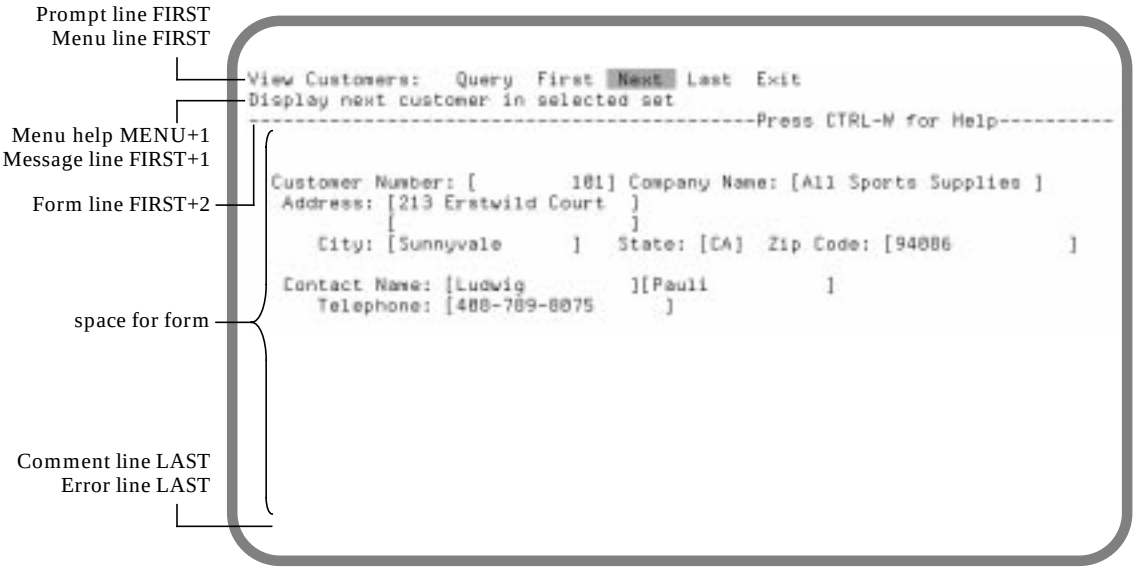
These statements are covered in detail under "Screen and Keyboard Options" on page 11-27 and in Chapter 3 of *INFORMIX-4GL Reference*.

Formatted Mode Display

In Formatted mode, certain lines of the two-dimensional screen are dedicated to certain types of information. You can change the location of these lines, but you cannot eliminate their special uses. Here are the reserved screen lines with their default positions:

Line Name	Purpose	Default Position	Point of Reference
Prompt	Output and input of PROMPT statement.	FIRST	4GL window
Menu	Ring menu display occupies two lines.	FIRST	4GL window
Message	Output of the MESSAGE statement.	FIRST+1	4GL window
Form	Top line of any form.	FIRST+2	4GL window
Comment	Explanatory comment for current form field.	LAST	4GL window
Error	Output of the ERROR statement.	LAST	4GL screen

Here is how the screen is arranged when the default line assignments are in effect.



To create this screen, a form was displayed using DISPLAY FORM; some data was written into form fields using DISPLAY…TO; and then a MENU statement was used to display a five option menu (the words “Query” through “Exit”). The line of dashes with “Press CONTROL-W for Help” is literal text in the first line of the form that was displayed.

This diagram makes the screen appear crowded with conflicting uses. In reality, the dedicated lines are used at distinct times. For example, the Prompt line is used by the PROMPT statement and the Menu line by the MENU statement. The program cannot execute both PROMPT and MENU at the same time, so no conflict is possible. By default, both lines are assigned to the first **4GL** window line.

The assignment of specific rows to screen lines can be changed while the program is running (see “Screen and Keyboard Options” on page 11-27). The key point is that these lines exist and have assigned display positions.

In the following example, screen output is achieved using DISPLAY AT, so the PROMPT statement uses only the current screen position (first line of the window, by default). As a result, this dialog will not scroll but will re-use the same two screen rows over and over.

```
MAIN
    DEFINE centDeg, fahrDeg DECIMAL(5,2)
    DEFINE keepOn CHAR(1)
    LET keepOn = "y"
    DISPLAY "Centigrade conversion" AT 12,1
    WHILE keepOn == "y"
        PROMPT "Centigrade temp: " FOR centDeg
        LET fahrDeg = (9*centDeg)/5 + 32
        DISPLAY centDeg, "C ==> ", fahrDeg, "F" AT 3,1
        PROMPT "More of this (y/n) ? " FOR CHAR keepOn
    END WHILE
END MAIN
```

Screens and Windows

In order to understand the way that forms and menus are used, you should understand the distinction between the **4GL** *screen* and a **4GL** *window*.

The Computer Screen and the 4GL Screen

The *computer screen* is the physical surface on which your program writes data. When **4GL** program output is directed to the screen, it appears in the **4GL** *screen*, also known as the *logical screen*.

If you are using a terminal, the entire computer screen is the **4GL** screen. If you are using a workstation, the window in which you are working is considered the **4GL** screen.

The 4GL Window

A **4GL** window is a rectangular area within which your program can display output. Initially, your program has one **4GL** window that fills the **4GL** screen. Additional **4GL** windows can be opened or closed as needed.

Each **4GL** window is a new rectangular area on which **4GL** can display output. You can use a secondary or subordinate **4GL** window the same way you use the first one: to display messages and to prompt for input, and to display menus and forms.

All **4GL** windows are contained inside the boundaries of the **4GL** screen. They can be the same size or smaller than the screen, but not larger. Any **4GL** window can overlap or completely obscure another **4GL** window.

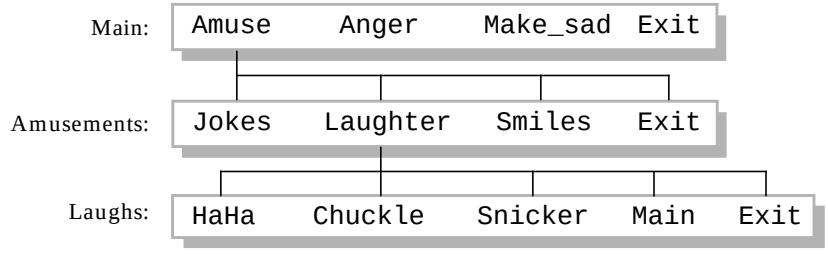
At any given time, a single **4GL** window is *current*. This is the window in which DISPLAY, MENU, and other interactive statements operate. Other windows can be completely or partly visible, but only the current window is active.

How Menus Are Used

The MENU statement allows you to offer the user a *ring menu* containing *menu options*. **4GL** displays the menu on the designated MENU line of the current window. As the user presses the TAB or arrow keys or the Spacebar, **4GL** moves the cursor from menu option to menu option. The user presses the RETURN key to select the current option. The user can also select a menu option by pressing the activating character (usually, but not always, set to be the first letter of the menu option) to select an option.

In your **4GL** program, you supply the list of options and, for each option, a block of statements. When the user selects an option, **4GL** executes the block of code corresponding to it. You can create as many levels of ring menus as you like.

One common use of this technique is to create nested menus.



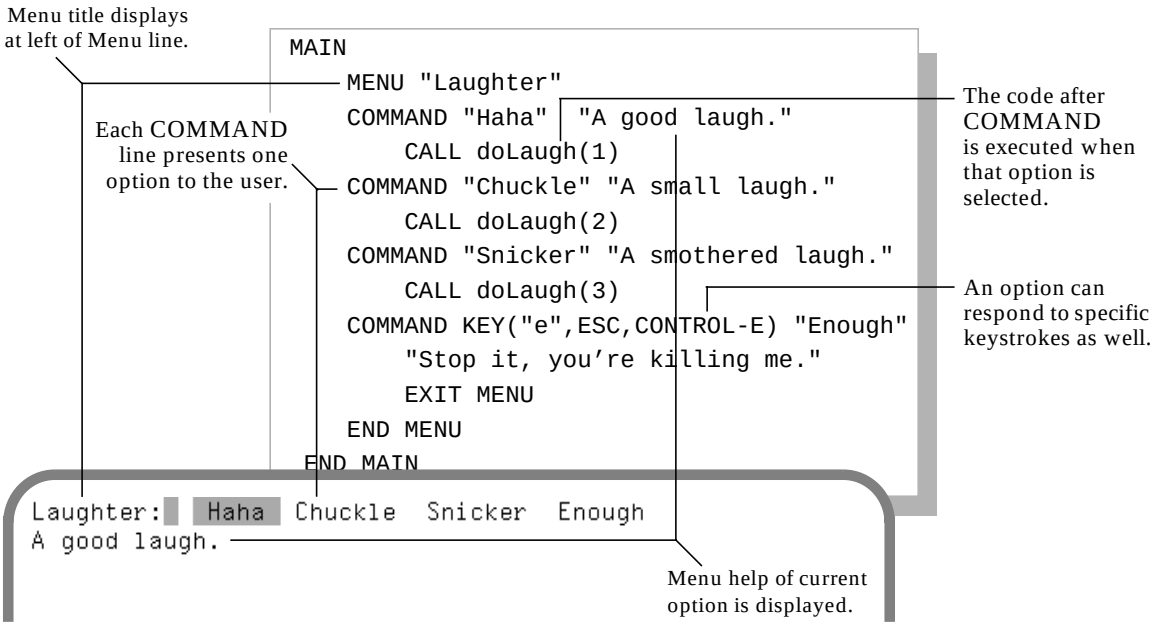
In the illustration, the initial menu offers at least three basic emotions. The user can choose to enter the Amuse, Anger, or Make_sad ring menu or Exit, to leave the program.

If the user chooses Amuse, the second tier ring menu replaces the first.

The Exit menu option moves you to the pervious menu or, if you are already at the top tier of that menu, exits the program. Alternatively, you could create a ring menu option that bypasses the natural hierarchial ring menu structure. An example is shown in the third tier menu, which offers the ability to jump back to the top tier by choosing Main.

There is no limit to the number of ring menu levels you can create.

Here is a simple menu-driven program. When it runs, the menu shown below it will be seen by the user.



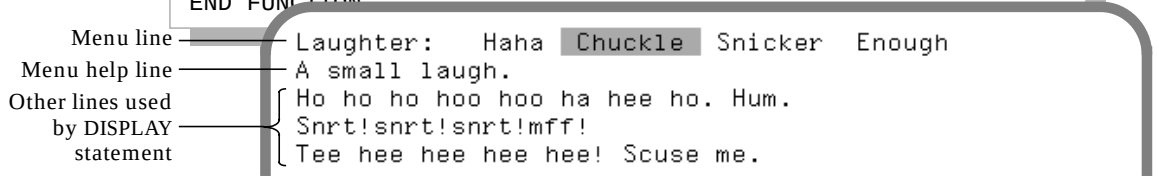
This program presents a menu containing four options: "Haha," "Chuckle," "Snicker," and "Enough." The user selects a menu option by typing its initial letter or by moving the cursor and pressing RETURN. You can also arrange for an option to be selected using other keys; for example, the option "Enough" can be chosen with the Escape key or the CONTROL-E key.

When the user selects a menu option, **4GL** executes the block of code that follows the COMMAND statement for that option. In this example, if the user selects the third option, "Snicker," the code `CALL doLaugh(3)` will be executed. When all the code for that option has been executed—in this case,

when the **doLaugh( )** function returns—the MENU statement resumes execution and the user can pick another option unless EXIT MENU is encountered, as in the case of "Enough."

Here is an implementation of **doLaugh()** and an example of the output it produced when several menu options were selected:

```
CONSTANT firstLaff = 3, lastLaff = 24
VARIABLE haha INTEGER = firstLaff
FUNCTION doLaugh(laffnum INTEGER)
    CASE laffnum
    WHEN 1
        DISPLAY "Ho ho ho hoo hoo ha hee ho. Hum." AT haha, 1
    WHEN 2
        DISPLAY "Tee hee hee hee hee! Scuse me." AT haha, 1
    WHEN 3
        DISPLAY "Snrt!snrt!snrt!mff!" AT haha, 1
    END CASE
    LET haha = haha + 1
    IF haha > lastLaff THEN LET haha = firstLaff END IF
END FUNCTION
```



After a command block completes, **4GL** redraws the Menu line and Menu help line, and waits for the user to choose another menu option. Program control remains within the MENU statement until it executes an EXIT MENU statement within some COMMAND block. The program on the previous page executes EXIT MENU when the "Enough" option is chosen.

You can write any number of lines of code in a command block. The example program shows a common style in which each command block contains a single function call. However, you can use most **4GL** statements in a command block. You can communicate with the user with MESSAGE, DISPLAY, or PROMPT statements; open additional **4GL** windows; or even start another MENU statement.

You can change the appearance of a menu while the program is executing. Within the menu, you can execute the HIDE and SHOW commands to hide or display menu options. For example, you could test the user's level of privilege in the current database, then either HIDE or SHOW a choice such as "delete row," depending on whether the user has Delete privilege.

All these features are covered in detail in the *INFORMIX-4GL Reference*.

How Forms Are Used

A form is a fixed arrangement of *fields* and *labels*. You design a form with fields to hold the data items you want to display, and *labels* to describe the *fields* to the user. Here is the form used in Example 11 in *INFORMIX-4GL by Example*.

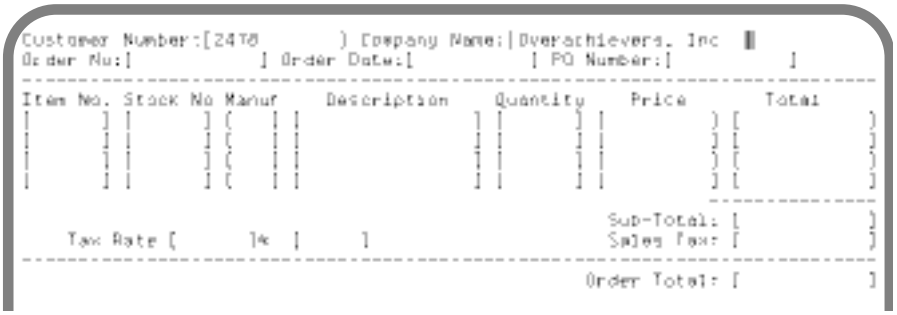


Figure 7-1 A sample screen form

Defining a Form

Two steps are involved in creating a form:

- Specify the contents of a form in a form specification file, an ASCII file you create with any text editor capable of generating ASCII text. Form specifications should be given the extension **.per**.
- Compile the form specification. Compiled forms are usually given the extension **.frm**.

The FORM4GL utility program is used to create **.frm** files. Once compiled, a **4GL** form can be used by any **4GL** program.

The form specification file has several sections. The DATABASE, SCREEN, and ATTRIBUTES sections are required, while the TABLES and INSTRUCTIONS sections are optional. The order of appearance of the sections is fixed. The sections of the form specification file are described next.

After you have designed a form and compiled the specification, it is ready for use by a program.

The form specification file that produces the form in Figure 7-1 on page 7-11 is considered in further detail beginning with "Specifying a Form" on page 11-3. Special syntax and keywords of form files are discussed in *INFORMIX-4GL Reference*, Chapter 5.

DATABASE Section

This section names a database from which column data types can be determined when the form is compiled. Alternatively, you can use the keyword FORMONLY to indicate that the form does not rely on a database. For example, you can identify the **stores2** database as follows:

```
DATABASE stores2
```

SCREEN Section

This section contains an ASCII version of the form, including text labels establishing the size and location of form fields. Here fields are labeled by *field tags*, internal names that are not displayed when the form appears at run time. This is also the SCREEN section of the form on page 7-11:

```
SCREEN
{

Customer Number:[f000       ] Company Name:[f001                ]
Order No:[f002       ] Order Date:[f003      ] PO Number:[f004      ]

Item No. Stock No Manuf    Description    Quantity    Price       Total
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]

}
END
```

TABLES Section

This section lists the tables or table aliases in the default database specified in the DATABASE section, or specified here by table qualifiers, from which column data types will be taken. An alias is required if an owner or database qualifier of the table name is needed. For example, you can identify tables as follows:

```
TABLES
customer
orders
items
stock
catalog
```

ATTRIBUTES Section

Here you specify the characteristics of each field: the field name, the type of data it will display, the editing rules applied during input, and any special display attributes such as color. For example:

```
ATTRIBUTES
f000 = customer.customer_num ;
f001 = customer.company ;
f002 = orders.order_num ;
f003 = orders.order_date ;
f004 = orders.po_num ;
f005 = items.item_num , NOENTRY ;
f006 = items.stock_num ;
f007 = items.manu_code ;
f008 = stock.description , NOENTRY ;
f009 = items.quantity ;
f010 = stock.unit_price , NOENTRY ;
f010 = items.total_price, NOENTRY ;
END
```

INSTRUCTIONS Section

In this section you specify fields in order to group fields into *screen records* and *screen arrays*. These records and arrays can be displayed and read as units.

```
INSTRUCTIONS
SCREEN RECORD s_items[4] ( item_num , stock_num , manu_code ,
    description , quantity, unit_price , total_price )
END
```

In the INSTRUCTIONS section, you can also change the default delimiters of form fields on character-based systems.

Displaying a Form

Your program uses a form in several ways:

- With OPEN FORM or OPEN WINDOW ... WITH FORM, you load the compiled form from disk into memory and make it ready for use.

  You can open as many forms as needed, subject only to the limits of memory and maximum number of open files on the platform you are using.
- With DISPLAY FORM, you draw the contents of a form (its labels and the outlines of its fields) in the current **4GL** window. The picture of the form replaces any previous data in that window.

  You can display a form as many times as necessary. You can display the same form into different **4GL** windows. (The use of additional windows is covered in "How 4GL Windows Are Used" on page 7-21.)
- With DISPLAY…TO, you fill the fields with data from program variables.

You can also use the CLEAR FORM statement to empty the fields of data.

Reading User Input from a Form

With the INPUT statement, your program waits for the user to supply data for specific fields of the form in the current **4GL** window. In the INPUT statement you list:

- The program variables that are to receive data from the form.
- The corresponding form fields that the user will use to supply the data.

When invoked, the INPUT statement enables the specified fields. The user moves the cursor from field to field and types new values. Each time the cursor leaves a field, the value typed into that field is deposited into the corresponding program variable. Other fields on the form are deactivated. The INPUT statement ends when the user does one of the following:

- Presses Accept (by default, ESCAPE) to resume execution and examine and process the values the user has entered.
- Presses Cancel (by default, CONTROL-C) to resume execution and ignore any changes made to the form.
- Completes entry of the last field, when field order is set to CONSTRAINED. This is the same as Accept. See "Field Order Constrained and Unconstrained" on page 11-26 as well as Chapters 3 and 5 of the *INFORMIX-4GL Reference*.

Screen Records

In the form file you can specify a group of fields as a logical screen record. During input, your program can associate a program record with a screen record, automatically filling the RECORD variable in memory with data from the form.

The main use of screen records is to make your program shorter and easier to read. You can use *asterisk* (wildcard) *notation* when referring to all the fields of the program record or the screen record. Suppose that your program defines a record variable as:

```
    DEFINE itemRow RECORD LIKE items.*
```

This creates a record variable with one member for each column of the **items** table. Now suppose that in the current form there are four fields that correspond to the last four columns of the **items** table (the schema for the stores demonstration database used in the following discussion is described in Appendix A of *INFORMIX-4GL Reference*).

In the form file, these fields are grouped into a screen record with the following line in the INSTRUCTIONS section:

```
INSTRUCTIONS
    SCREEN RECORD itemDetail(stock_num, manu_code, quantity,
                         total_price)
```

The program could take input from the four fields by specifying the fields and the corresponding record members individually:

```
INPUT   itemRow.stock_num, itemRow.manu_code,
            item_row.quantity, itemRow.total_price
FROM    stock_num, manu_code, quantity, total_price
```

Or you can specify the last four members of the program record using THRU notation and all the fields of the screen record using an asterisk:

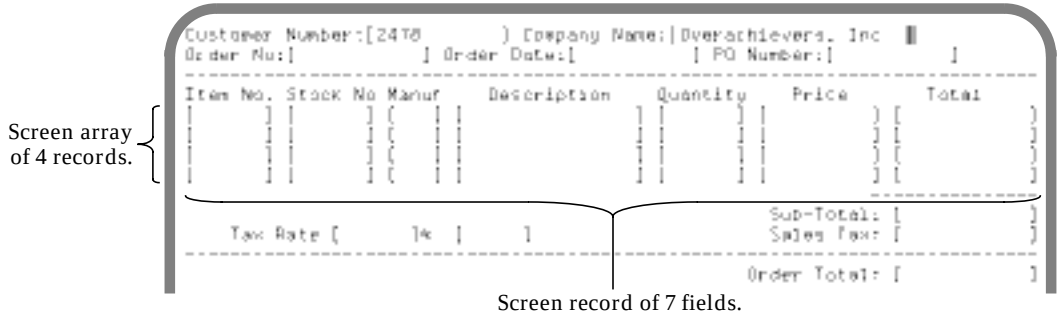
```
INPUT   itemRow.stock_num THRU itemRow.total_price
    FROM itemDetail.*
```

But since the names of the members in the program record are the same as the names of the form fields, this can be further shortened to:

```
INPUT BY NAME itemRow.stock_num THRU itemRow.total_price
```

Screen Arrays

In the form specification, you can also specify a group of screen records as a *screen array.* During input you can associate a *program array* of records with an array of form fields on the screen.



Typically the program array has many more rows of data than will fit on the screen.

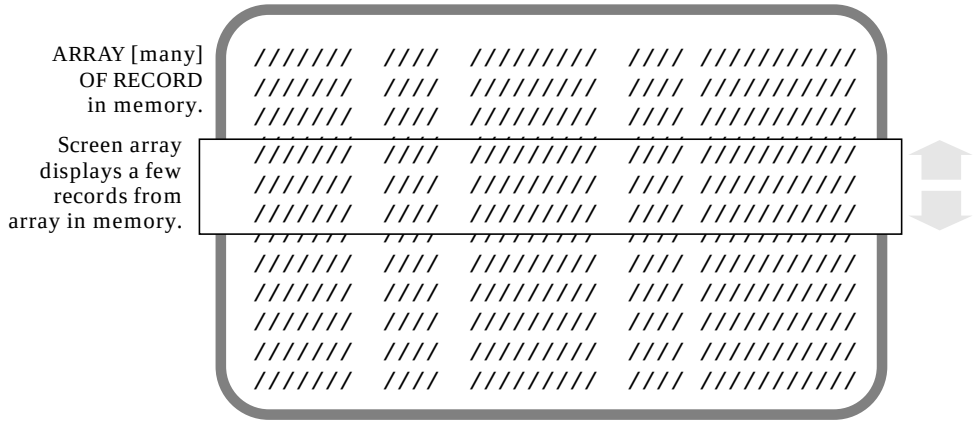


Figure 7-2 The typical screen array is part of a much larger program array

4GL lets the user *scroll* the array on the screen through the rows of the program array. The user can change the display using the **4GL** PageUp and PageDown logical keys or scroll through the array one line at a time using the arrow keys.

To add a record, the user can press the logical Insert key, and **4GL** will open the display to create an empty screen record. When the user has filled this record, **4GL** inserts the data into the program array.

To delete data, the user can press the logical Delete key, and **4GL** will delete the current record from the display and from the program array, and redraw the screen array so that deleted records will no longer be shown. Depending on how your program is written, you can also programmatically remove the record from the database. For a complete list of logical key assignments, see the description of the OPTIONS statement in *INFORMIX-4GL Reference* or see the *INFORMIX-4GL Quick Syntax*.

How the Input Process Is Controlled

Your program stops in the INPUT statement and waits while the user enters data. However, you can write blocks of code that are automatically called by **4GL** during input, so that you can monitor and control the actions of your user within this statement. You can use the following statements in conjunction with INPUT:

BEFORE INPUT	Just as the INPUT operation is starting, this block of code can display initial or default values, clear totals, and generally prepare the screen and program variables.
BEFORE FIELD	As the cursor is entering the specified field, this block can initialize the field contents based on values in other fields.
AFTER FIELD	When input in the specified field is complete, this block can validate the user's input, or can initialize other fields based on the value just entered.
ON KEY	When the user presses any of a list of keys you specify, this block can give the user special assistance, for example, displaying a list of common values for the current field.
BEFORE ROW	When the cursor is about to enter a new row of a screen array, this block can update other fields on the screen to reflect the row being entered.
AFTER ROW	When the cursor is leaving a row of a screen array, this block can update the screen or the database to account for changes made in the fields of the row.
BEFORE INSERT	When the user has requested creation of a new row in a screen array, this block can initialize the new row with default values.
AFTER INSERT	When the cursor is about to leave a newly-inserted row of a screen array, this block can update totals based on the new data, and can insert the new row into the database.
AFTER INPUT	When the input operation is ending, this block can validate the entered data, check for required fields that may be missing, and erase any special usage messages.

You write these blocks as part of the INPUT statement. When the INPUT statement is executed, **4GL** enables the screen for input and awaits user keystrokes. When the user presses a key that creates one of the situations above, **4GL** automatically calls your block of code.

You can use any **4GL** statements in these blocks, as well as two special statements that can control the display.

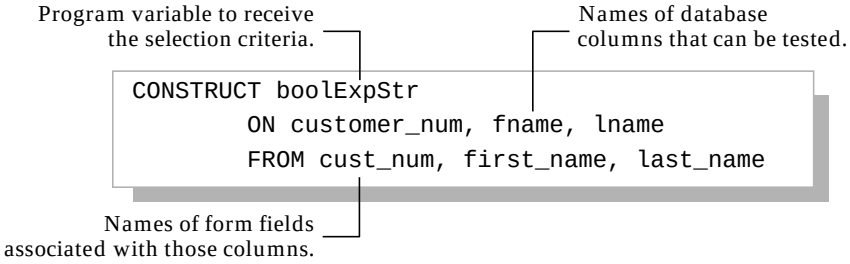
NEXT FIELD	Used in an AFTER FIELD or ON KEY block to direct the cursor to a specified next field, or back to the same field to correct an error in data entry.
NEXT ROW	Used in an AFTER ROW or ON KEY block to move the cursor to a particular row of a screen array. **4GL** scrolls the array as necessary to show the specified row.

NEXT FIELD and NEXT ROW should only be used in situations where the program controls the order that fields are visited by the user. Please see "Field Order Constrained and Unconstrained" on page 11-26 for additional information on programming this type of user interaction.

How Query by Example Is Done

4GL lets you take input from a form in another way: instead of literal values for the program to process, your user can enter relational tests for a query. The process is known as *Query by Example*. The user enters a value or a range of values for one or several form fields. Then your program looks up the database rows that satisfy the requirements.

The **4GL** statement that makes Query by Example possible is CONSTRUCT.



The CONSTRUCT statement operates very much like INPUT. In it you list names of database columns and names of fields in the current form that correspond to those columns.

You provide a single program variable to hold the result of the query. When the CONSTRUCT statement executes, **4GL** enables the form fields listed in this CONSTRUCT statement for input. The user can enter either specific values, or requirements such as >5 (meaning any value greater than 5) or ="Sm[iy]th*" (meaning any value beginning with "Smith" or "Smyth").

When the user presses Accept, **4GL** converts the input into a Boolean expression suitable for use in the WHERE clause of a SELECT statement. This character string is returned to the program variable. When the CONSTRUCT statement shown in the previous illustration has completed, the following character string might be stored in the program variable **boolExpStr**:

```
customer_num > 5 AND lname MATCHES "Sm[iy]th*"
```

Now it is up to your program to use the Boolean expression to fetch the database row, or rows, that the user wants to see. The steps include:

- Combine the Boolean expression string with other text to form a complete SELECT statement.

```
LET fullStmt = "SELECT * FROM customer WHERE " , boolExpStr
```

- Prepare the SELECT statement for execution.

```
PREPARE qbeSel FROM fullStmt
```

- Associate the prepared statement with a database cursor.

```
DECLARE qbeCur CURSOR FOR qbeSel
```

- Open the cursor and fetch the row(s) it has selected.

What you do with the rows depends on the specific application. Often the reason for the CONSTRUCT is to select rows to be viewed by the user. In such a case, the program could display each row individually in a form or grouped in a screen array. Or you might choose a set of rows for a report, or a set of rows to be deleted or updated, and so forth.

The CONSTRUCT statement supports the same features of BEFORE FIELD and AFTER FIELD program blocks and ON KEY blocks as the INPUT statement.

How 4GL Windows Are Used

Each **4GL** program begins with a **4GL** window that fills and covers the entire **4GL** screen. This screen area may be:

- A terminal-emulation window on a workstation
- The physical screen of a terminal

Additional **4GL** windows can be created and further manipulated with the following statements:

Statement	Purpose
OPEN WINDOW	Create a new window and make it the current window. You specify its size and location in relation to the upper-left corner of the screen.
CLOSE WINDOW	Close and discard a window by name.
CLEAR WINDOW	Empty the contents of a window, erasing anything displayed on it.
CURRENT WINDOW	Bring a window to the front if necessary, and make it current.

Only the current **4GL** window has keyboard focus. This means that any interaction initiated by the user through the keyboard goes to the current **4GL** window.

The statements listed on page 7-5 all operate upon the current **4GL** window.

Alerts and Modal Dialogs

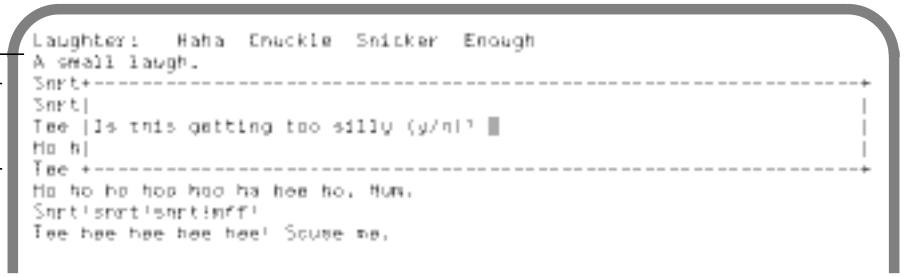
One frequent use of a **4GL** window is to display an error message or a dialog box that the user must respond to. With a few statements you can:

- Open a **4GL** window.
- Conduct a dialog with the user.
- Close the **4GL** window.

Here is what an alert window might look like:

Main **4GL** window being used by MENU and DISPLAY.

Subordinate **4GL** window being used for dialog via PROMPT.



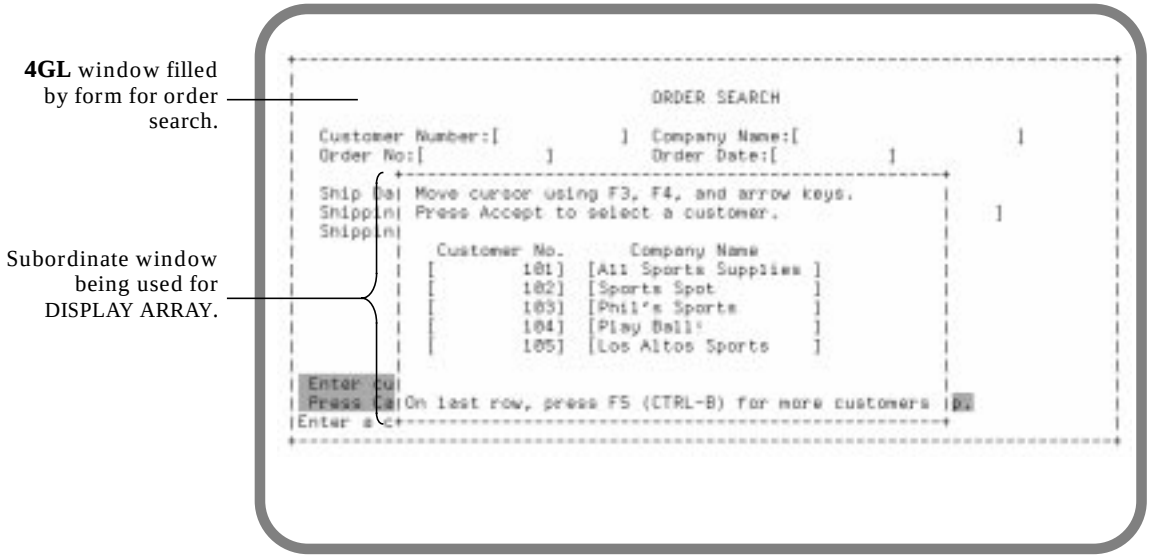
The code that presented this window follows:

```
FUNCTION domodal()
    DEFINE ansr CHAR(1)
    OPEN WINDOW modal AT 4,6
        WITH 3 ROWS, 64 COLUMNS
        ATTRIBUTE(BORDER, PROMPT LINE 2)
    PROMPT "Is this getting too silly (y/n)? " FOR CHAR ansr
    CLOSE WINDOW modal
    RETURN ( “Y” = UPSHIFT(ansr) )
END FUNCTION
```

The **domodal( )** function opens a subordinate **4GL** window, prompts the user for a single-letter response, closes the window, and returns either TRUE or FALSE depending on whether the response was the letter “Y” or not. A function like this can be used at almost any point within a program.

Information Displays

Another common use for a **4GL** window is to display helpful information to the user during input. In the following illustration you see a larger window containing a form, partly covered by another **4GL** window.



The user is to enter a customer number in a field of the form. The user pressed a designated key to ask for help. Within the INPUT statement, in the ON KEY block for that key, the program has called a function that does the following:

- Opened a **4GL** window.
- Displayed a form in that **4GL** window.
- Used DISPLAY ARRAY to show a scrolling list of rows from the **customer** table, using the form in the current **4GL** window.

When the user presses Accept, the function will note the customer number in the last-current row. This will be returned as the result of the function. Before the function returns, it will close the **4GL** window it opened. That makes the larger **4GL** window current again.

The ON KEY block will display the returned customer number into the form field, so the user will not have to enter it.

How the Help System Works

4GL supports a simple and effective system of providing help to users in the form of help messages associated with interactive statements in your program. The purpose of a help message is to guide the user when it may not be clear how you meant the program to be used.

Here are the typical steps you might take in creating a useful help facility for an application:

- Think of each situation where the user might need detailed guidance, and write a message explaining what to do at that point.

  The text can be as long as you wish. Only printable characters, spaces, and tabs are allowed.
- Assign to each message a unique positive integer number between 1 and 32700.
- Put the message texts with their numbers and optional comments into a text file.
- Compile the text file using the **4GL** message compiler utility **mkmessage**.
- In your program source file, specify the name of the file of compiled help messages in an OPTIONS HELP FILE statement.
- In each interactive statement, specify the HELP *number* clause to name the help message number that you designed for that statement.

If the user presses the Help key (CONTROL-W by default) while the interactive statement is executing, **4GL** displays the message whose number you specified.

The display takes up the full current window, hiding the main and subordinate windows.

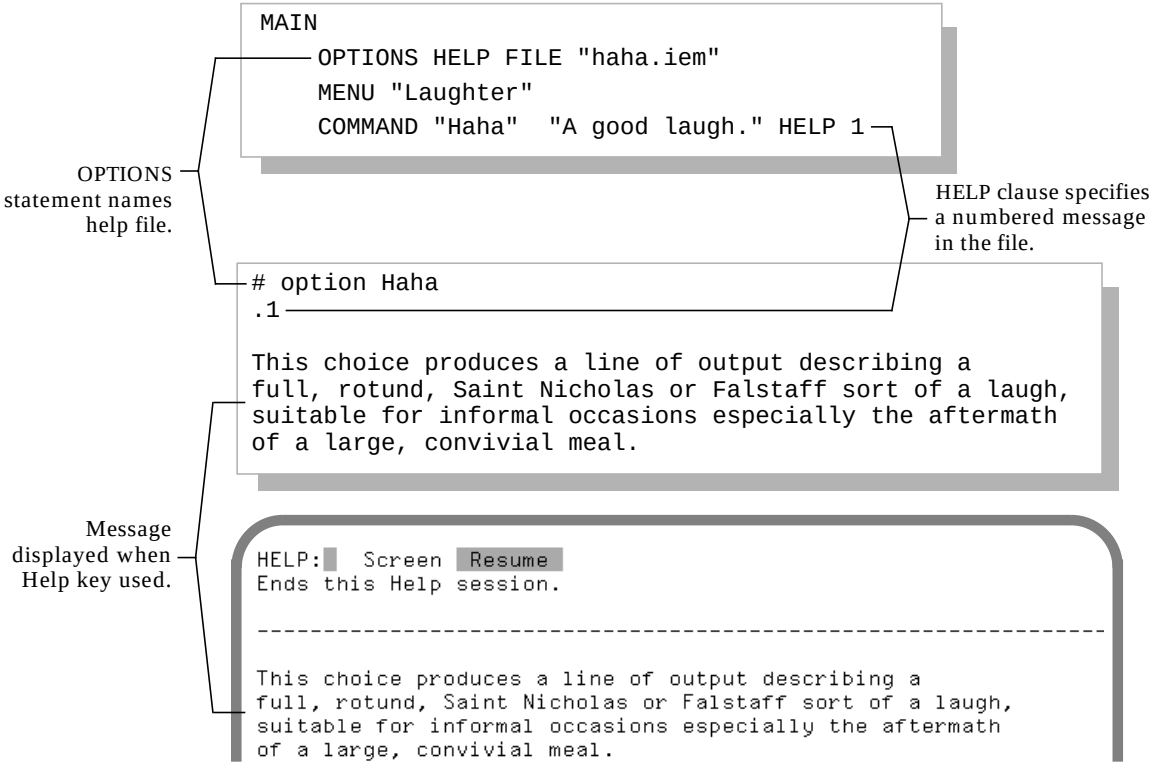


Figure 7-3 A source module, an ASCII help file, and a help window

Like a form file, a message file can be used with more than one program. You can use more than one message file within a program; executing the OPTIONS HELP FILE statement changes the active file at any time.

You can write a HELP *number* clause in any statement that requests input from the user: PROMPT, INPUT, or CONSTRUCT. You can specify a different help message number for each option of a MENU statement. In addition, you can start the help display from anywhere in the program by calling the library function **showhelp(*n*)**. Using this function you can start a help display from, for example, an ON KEY block of a DISPLAY statement.

Chapter

8

Using the Language

Overview

INFORMIX-4GL has the features of a structured language such as Pascal or C as well as advanced features of its own. This section surveys **4GL** as a programming language.

Simple Data Types

4GL supports a number of *simple data types*. They are called "simple" because each type describes a single item of information (as opposed to a collection or *array* of items). The simple data types of **4GL** are a robust subset of the data types that you can use as database columns with Informix database engines.

Each of the simple types is discussed in detail in *INFORMIX-4GL Reference*, Chapter 3. There you will find information on their minimum and maximum capacities, the proper format for literal values of each type, and other details. The types are summarized here as background material for the rest of the chapter.

Number Data Types

4GL supports seven different representations of numeric values (some of which have more than one keyword).

Datatype Keywords	Kind of Data Represented
DEC(*p*, *s*) DECIMAL(*p*, *s*) NUMERIC(*p*,*s*)	Decimal fixed-point numbers with specified precision and scale.
MONEY(*p*, *s*), MONEY(*p*), MONEY	Currency amounts with specified precision and scale (defaulting to 16, 2).
DEC(*p*) DECIMAL(*p*) NUMERIC(*p*)	Decimal floating-point numbers with specified precision (defaulting to 16).
FLOAT DOUBLE PRECISION	Binary floating-point numbers with precision of C *double*.
REAL SMALLFLOAT	Binary floating-point numbers with precision of C *float*.
INT INTEGER	Whole numbers, from -2,147,483,647 to +2,147,483,647.
SMALLINT	Whole numbers from -32,767 to +32,767.

Synonyms such as REAL for SMALLFLOAT are supported to conform to the ANSI standard for SQL.

Differences Between DECIMAL and MONEY Data Types

The internal representations of fixed DECIMAL and MONEY are identical; the only difference between them is that when **4GL** displays a MONEY value to the screen or a report, it formats the number as currency.

Numeric Precision

Some **4GL** number types are implemented as C standard data types, as follows:

FLOAT	Same as C double
SMALLFLOAT	Same as C float
INTEGER	Same as C long
SMALLINT	Same as C short

A library of C functions for working with DECIMAL and MONEY data types is included with both **4GL** and the **INFORMIX-ESQL/C** product.

Time Data Types

4GL supports three data types for keeping track of time.

Datatype	Kind of Data Represented
DATE	Points in time specified as calendar dates.
DATETIME	Points in time stored with a specified precision, as calendar dates and / or times of day.
INTERVAL	Spans of time stored with a specified precision, either years and months, or in days and hours.

A DATE value is stored as a count of days before or after midnight, 31 December 1899; that is, January 1, 1900 would be stored as zero. When it displays a DATE value to the screen or a report, **4GL** formats it according to directions in the DBDATE environment variable, so your user can tailor the display of date values to match local conventions. (You can also employ the USING operator to format DATE values.)

You can mix DATE values with integers when doing arithmetic (for example, subtracting 7 to get a date for the same day of the previous week). In addition, a number of built-in operators accept or return DATE values. Built-in functions and operators are described in Chapter 4 of the *INFORMIX-4GL Reference*.

Built-in Operator	Purpose
DAY(*date-expr*)	Return number of the day of the month from a DATE or DATETIME.
MONTH(*date-expr*)	Return number of the month from a DATE or DATETIME.
WEEKDAY(*date-expr*)	Return number of the day of the week from a DATE or DATETIME.
EXTEND(*date*, *qual*)	Changes the precision of a DATE or DATETIME value, returning a DATETIME value.
YEAR(*date-expr*)	Return number of the year from a DATE or DATETIME.
DATE(*expr*)	Convert integer or character string to DATE or DATETIME.
MDY(*m*, *d*, *y*)	Compose a DATE from integer values for month, day, year.
TODAY	An expression yielding the current date as a DATE.

These operators are identical in name and use to functions available in SQL statements. Used in SQL, they apply to values in the database. You can also use them in **4GL** statements, applying them to program variables.

A DATETIME value can have more or less precision than a DATE: it can specify a date and/or a time, and can be exact to a fraction of a second.

An INTERVAL represents a span of time, not a particular moment in time. For example, "three hours" is an interval; "three o'clock" is a point in time. You can do arithmetic that mixes DATE, DATETIME, and INTERVAL values, yielding new DATETIME or INTERVAL values.

There are detailed discussions of the DATETIME and INTERVAL data types in the *INFORMIX-4GL Reference*.

Character and String Types

4GL supports four ways to represent strings of bytes in memory, with different abilities and uses.

Datatype	Kind of Data Represented
CHAR(*length*) CHARACTER(*length*)	Character strings of fixed length up to 32,767 ASCII characters.
VARCHAR(*length*)	Character strings of varying length, up to 255 bytes.
TEXT	Character strings up to 2^{31} bytes.

The most important of these is CHAR or its synonym CHARACTER, which is the type of most character strings in a typical program. In fact, CHAR is the default **4GL** datatype. The following built-in operators accept or return character values (either CHAR or VARCHAR).

Operator	Purpose
char-variable [*start*, *end*]	Select a substring from a CHAR or VARCHAR value.
ASCII *int-expr*	Return a specific ASCII character as a CHAR(1) value.
char-expr CLIPPED	Return a character value stripped of any trailing spaces.
LENGTH(*char-expr*)	Return length of character value exclusive of trailing spaces.
value USING *"pattern"*	Return character representation of a value, formatted to fit a pattern.

Within a DISPLAY statement (that displays values to the screen) or a PRINT statement (that sends values to a report), you can use the COLUMN operator to set the beginning position of the value of the current line of output.

CHAR and VARCHAR Compared

In the database, the important difference between CHAR and VARCHAR columns is that only the actual length of a VARCHAR value is stored to disk, while a CHAR value always occupies its full declared size.

This difference is not important for program variables because **4GL** always allocates enough memory to hold the defined size of a VARCHAR value. For example, if you define a VARCHAR(25), it always occupies 26 bytes of memory even if you store a 1 byte value in it. Thus you cannot economize on program memory by using VARCHAR in place of CHAR.

In a program, the difference between the two types is that when you reference a CHAR variable, you always get its full size in characters, filled out with railing spaces if necessary. The CLIPPED operator ban be used to drop these trailing spaces. When you refer to a VARCHAR variable, you get only its current contents without any padding. Hence the CLIPPED operator is less often needed with VARCHAR values in the program.

Binary Large Objects

The types BYTE and TEXT are collectively known as *blob* (Binary Large Object) data types; they represent strings of data that can be of any length. The only difference between a BYTE and a TEXT datatype is that a BYTE value can contain any combination of binary values, while TEXT data items contain any number of ASCII values.

INFORMIX-OnLine Dynamic Server<Default ¶ Fo> database engines support blob data types.

In **4GL** you can use the LOCATE statement to specify whether the contents of a blob variable are to be held in memory or in a disk file, and you can change this location dynamically, as the program runs.

The main uses for blob variables are:

- To fetch them from the database into program variables of the same type.

  When the receiving variable is located in a file, this is effectively a disk-to-disk copy from the database to the file.

- To store them from program variables into the database.

  When the source variable is located in a file, this is effectively a disk-to-disk copy from the file to the database.

- To view, or even modify, a blob value using any external program that understands the value's contents. An example of this would be retrieving a blob consisting of a graphic from an INFORMIX-OnLine database,

calling a "paint-type" editing program, making changes in the graphic, and reloading the modified graphic in the database after the revised image had been saved. Obviously modifications of graphic images generally requires a graphical user interface.

Variables and Data Structures

A program *variable* is a named location in memory where a value can be stored. There are four things to know about any variable:

- Its *type*; that is, what type of data can it hold?

  A variable can hold a particular type of data.

- Its *structure*; that is, does it contain only a single value or is it a collection of multiple values? If it is an *aggregate*, how are the individual simple values accessed?

  For example, an array is a collection of values of the same type. You access a single value by writing a subscript, as in **custNumList[15]**.

- Its *scope of reference*; that is, in what parts of the program can you use its name?

  There are three possibilities discussed in greater detail later in this chapter; they are *local*, *module*, and *global*.

- Its *time of allocation*; that is, when during the execution of the program is it created and initialized?

  There are two possible times: the variable can be allocated at compile time as part of the executable program file, or it can be allocated at run-time, dynamically, while the program is running.

Declaring the Data Type

All four characteristics of a variable are decided when you *declare* a variable. To declare a variable is to write a statement that tells **4GL** about the variable. The keyword DEFINE is used for this. Here is a DEFINE statement that declares four simple variables.

```
DEFINE
    j, custNum INTEGER ,
    callDate DATETIME YEAR TO SECOND ,
    sorryMsg CHAR(40)
```

You state the *type* of a variable after its name. In the preceding declaration, the types of variables **j** and **custNum** are INTEGER; the type of **callDate** is DATETIME YEAR TO SECOND, and the type of the variable called **sorryMsg** is CHAR(40). Any of the simple types listed earlier in this chapter can be used.

You can also use the LIKE keyword to specify that the type of a variable is the same as the type of a specified column in a database.

```
DEFINE custFname LIKE customer.fname
```

The advantage of LIKE is that if the database schema changes, you need only recompile your program to make sure that the datatypes of your variables match those in the database.

Creating Structured Datatypes

Until now only simple datatypes have been considered. **4GL** also supports data types that contain many individual data items. Such aggregate data types are considered to have a structure. You specify the structure of a variable by stating that it is an ARRAY or a RECORD in the DEFINE statement.

Declaring an Array

An *array* is a collection of elements all of the same type, ordered along one or more dimensions. Here are two examples of array declarations:

```
DEFINE custNumTab ARRAY [2000] OF LIKE customer.customer_num
DEFINE custByProd ARRAY [100, 25] OF MONEY(12)
```

The number of elements is specified in brackets. The example shows a single dimension array 2,000 elements long and a 100 by 25 (100*x*25) two-dimensional array. Three-dimensional arrays can also be created.

All elements of an array have the same data type that you can specify using the OF clause. The type can be one of the simple types or can be a record.

You access an element of an array by naming the array with a subscript expression in brackets, as in `custNumTab[175]` or `custByProd[j,prodNum]`.

Declaring a Record

A record is a collection of variables, each with its own data type and name. You put them in a record so you can treat them as a group.

```
DEFINE person RECORD
    honorific VARCHAR(40) , -- e.g. "Excellency"
    initial CHAR(1) ,
    famName CHAR(30)
END RECORD
```

You access an member of a record by writing the name of the record, a dot (known as *dot notation*), and the name of the member; for example, **person.initial** is the second member of the **person** record.

You can declare a record that has one member for each column in a database table. The names of the members and their data types are derived from the database. The only exception is that SERIAL data types are converted to INTEGER data types. In the simplest form, you write:

```
    RECORD LIKE tablename.*
```

As in the following:

```
    DEFINE custRec RECORD LIKE customer.*
```

The statement creates a record named **custRec** having one member for each column of the **customer** table. Each record member has the name and the data type of the corresponding column in the table.

You can augment table columns with other members; the clause:

```
 LIKE tablename.*
```

retrieves the names of columns and their types. This, in effect, defines a RECORD.

```
DEFINE custPlus RECORD
        row INTEGER ,
        customer RECORD LIKE customer.* ,
        balanceDue DECIMAL(8,2)
    END RECORD
```

The preceding statement creates a record named **custPlus** having a member for each column of the **customer** table, and two additional members. The member **custPlus.row** is an integer. The member **custPlus.balanceDue** is a decimal number. In this case, where the LIKE clause only generates some of the members of the record, you must use an END RECORD clause to finish the record definition.

Since **custPlus.customer** is a record within the record, a reference to the **lname** member of the record is specified as:

```
    custplus.customer.lname
```

Declaring the Scope of a Variable

You specify the scope of a variable within its source module by where in the source module you write the DEFINE statement. If the DEFINE statement is:

- Within a function, the scope is local to that function. The variable can only be referenced while the function is executing. (Functions are described on page 8-29.)
- At the top of the source module and outside any MAIN, REPORT, or FUNCTION block, the variable is considered a *module variable*; its name can be used anywhere from that point to the end of the source module.
- In a GLOBALS statement in a module separate from any other statements, the variable is available in that module (and in any other module that includes the GLOBALS *filename.4gl* statement that defines that variable).

The context of the DEFINE statement also determines the following characteristics of a variable:

- Scope of reference—where the **4GL** compiler recognizes the variable name
- Time of allocation—when **4GL** allocates memory for the variable

4GL also supports recursion (a function calling itself). Each separate call to a function allocates its own copy of local variables.

Scope of Variable Reference

The context of a DEFINE statement determines the *scope of reference* of a variable, often referred to simply as *scope*. During compile-time, scope is that portion of the source code in which the **4GL** compiler can recognize a particular variable name. Outside its scope, the variable name is unknown or may even be defined differently.

Local Scope: Within a Program Block

Within the definition of a function, or within a MAIN or REPORT program block, DEFINE creates *local* variables. The scope of reference of a local variable is restricted to the same program block. The DEFINE statements that declare local variables must precede any executable statement within the same program block.

Modular Scope: Within a Source Module

Outside any FUNCTION, REPORT, or MAIN program block, DEFINE creates *module* variables. The scope of reference of a module variable is the entire source module. Module variable definitions must appear at the top of the source module, before any executable statements.

Global Scope: Within Several Modules

The GLOBALS *filename.4gl* statement can extend the scope of module variables that you define outside any FUNCTION, REPORT, or MAIN program block and within a GLOBALS...END GLOBALS statement. The scope of reference for a global variable is:

- The entire module in which the GLOBALS...END GLOBALS statement appears.
- Any other modules that contain GLOBALS *filename.4gl* statement.

Both the GLOBALS...END GLOBALS and the GLOBALS *filename.4gl* statements must appear at the top of a source module, before any executable statements.

Time of Variable Allocation

The context of the DEFINE statement also determines when the memory for the variable is allocated. **4GL** handles memory allocation differently for the different variable scopes.

Allocation of Local Variables

Storage for local variables is allocated *dynamically*. **4GL** allocates this storage when the program block (MAIN, FUNCTION, or REPORT statement) containing the variable begins execution.

Local variables are initialized in the same order that their names appear in the program block.

Allocation of Module Variables

Storage for module variables is allocated statically, in the executable image of the program. **4GL** allocates this storage when the program begins execution and deallocates it when the program exits.

Module variables are initialized in the same order that their names (and the names of global variables) appear in the source module.

Allocation of Global Variables

Storage for global variables is also allocated statically. **4GL** allocates this storage when the program begins execution and deallocates it when the program exits. However, to be able to handle references to a global variable across several source modules, **4GL** makes a distinction between:

- *Variable declaration:* tells the **4GL** compiler the name and the data type of the variables so that the compiler can verify references to this variable in a given source module.
- *Variable definition:* allocates the memory for the global variable. For global variables, memory is allocated statically, as part of the program image.

The GLOBALS...END GLOBALS statement *defines* the global variables. It also *declares* them so the compiler can verify references to a global variable in the same module it is defined.

To make a global variable visible in other modules of the program, you only need to declare these variables. You do not need to define them because memory only needs to be allocated once and is done so with the GLOBALS...END GLOBALS statement.

To declare global variables in other modules, you must:

- Put the GLOBALS...END GLOBALS statement in a separate **4GL** source file.
- At the top of each source file which references a global variable, put the GLOBALS *filename.4gl* statement, where *filename.4gl* is the name of a file containing the GLOBALS...END GLOBALS statement.

The following declares global variables in the **globs.4gl** source file:

```
GLOBALS
    DEFINE a,b,c INT ,
    x,y,z CHAR(10)
END GLOBALS
```

To reference these variables in other source modules, you would put the following statement at the top of each source module using a global variable:

```
GLOBALS "globs.4gl"
```

The following section discusses global variable declarations in detail.

Using Global Variables

The GLOBALS statement defines and declares a global variable. All references to a global variable refer to the same location in memory. This statement has two forms:

- If you use the LIKE keyword in any DEFINE statement, you must identify the database that contains the referenced database columns. You can do this in one of two ways:
  - Precede the GLOBALS...END GLOBALS with a DATABASE statement specifying the database containing the referenced columns
  - Qualify each referenced column with its table name
- If a global variable defined within the GLOBALS...END GLOBALS block has the same name as a local variable, then the local identifier takes precedence within its scope of reference. A module variable in the same source module *cannot* have the same name as a global variable.

The following program segment defines a variable like the **customer** table of the **stores2** demonstration database:

```
DATABASE stores2
GLOBALS
    DEFINE p_customer RECORD LIKE customer.*
END GLOBALS
```

Global Variable Declaration

The *globals* file contains global variable definitions and must have a **.4gl** file extension; for example, this file may be named **globals.4gl**. You compile this file as part of your multiple-module program. This globals file should contain *only* GLOBALS...END GLOBALS blocks, no **4GL** executable statements (DATABASE and DEFINE statements are valid, however). Different source files can reference different globals files.

Using GLOBALS Within a Single Module

The following program fragment defines a global record, a global array, and a simple global variable that are referenced by code in the same source module:

```
DATABASE stores2
GLOBALS
DEFINE p_customer RECORD LIKE lbraz.customer.* ,
    p_state ARRAY[50] OF RECORD LIKE state.* ,
    state_cnt SMALLINT ,
    arraysize SMALLINT
END GLOBALS

MAIN

   LET arraysize = 50
 ...
END MAIN

FUNCTION get_states()
 ...
FOREACH c_state INTO p_state[state_cnt].*
  LET state_cnt = state_cnt + 1
  IF state_cnt > arraysize THEN
    EXIT FOREACH
  END IF
END FOREACH
 ...
END FUNCTION
```

```
FUNCTION state_help()
DEFINE idx SMALLINT ...
CALL SET_COUNT(state_cnt)
DISPLAY ARRAY p_state TO s_state.*
LET idx = ARR_CURR()
LET p_customer.state = p_state[idx].code
DISPLAY BY NAME p_customer.state
 ...
END FUNCTION
```

The **INFORMIX-4GL** compiler will generate an error if you defined a module variable with a name of **arraysize**, **p_customer**, **p_state**, or **state_cnt** in the same module containing the GLOBALS statement.

However, used in this context, the GLOBALS statement simply defines variables with module scope. For other modules to be able to access the variables, they must use the GLOBALS *filename* statement and to do so, the GLOBALS...END GLOBALS block must appear in a separate source file.

Global Versus Module Scope

When a variable is declared in a GLOBALS statement, the variable is still a module variable as far as the current source module is concerned, but you can use it anywhere in your program. When you do this, the source modules share the single copy of the variable. If the value is changed in one module, it is changed through the program.

Note: *You cannot specify GLOBALS "filename" if the file being referenced contains any executable statements such as LET.*

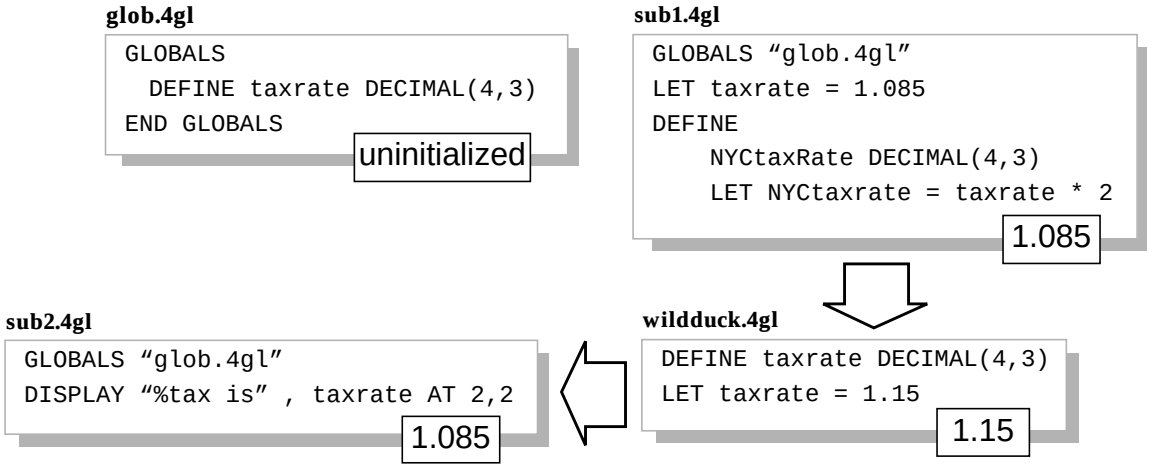


Figure 8-1 ***Illustrates the difference between global and local variables.*** *The numbers in boxes indicate the value of taxRate at the particular point in the program.*

In the preceding illustration, the source module **glob.4gl** declares a variable **taxrate** as a global variable. The current value of the global variable is available to any function in any module that references the global source module.

Another source module, **sub1.4gl**, references the GLOBALS file **glob.4gl**, and assigns a value of `1.085` to **taxrate**.

A third module, **wildduck.4gl**, defines a module variable named **taxrate** and assigns it a value. Since there is no reference to the globals file **glob.4gl**, this module-level assignment is permitted. No change in the value of **taxRate** in **wildDuck.4gl** will affect the current value of the global **taxRate**.

Finally, a routine in a fourth module, **sub2.4gl**, displays the current value of the global variable **taxrate**, `1.085`.

Initializing Variables

You initialize variables using the LET statement.

```
DEFINE Pi, TwoPi SMALLFLOAT
    LET Pi = 3.1415926
    LET TwoPi = 2*Pi
```

The declaration of a variable must precede any executable statement in the same program block.

4GL programmers often initialize global variables at a single point in the program, such as immediately after the last DEFINE or DATABASE statement in the MAIN program block.

Expressions and Values

A *value* is a specific item of information. An *expression* specifies how a value is to be produced. You can use expressions at many points in a **4GL** program, for example in:

- Function calls, to specify the arguments to functions
- Decision and looping statements, to control program flow
- Reports, to specify the values to print in the report
- SQL statements, to specify values to be inserted into the database
- LET statements, to specify values to be assigned to variables

Every value has a type, one of the simple data types listed earlier in this chapter. The **4GL** compiler knows the type of every part of an expression, so it can tell the type of the final value that the expression describes.

See the *INFORMIX-4GL Reference* for a detailed discussion of **4GL** and SQL expressions.

Literal Values

The simplest kind of expression is a *literal* value, that is, a literal representation of a number, character string, or other kind of data.

Here are some literal numbers: `-7`, `3.1415926`, `1e-8`. **4GL** assumes any literal number with a fractional component has the type DECIMAL(32), the most precise number type available.

You write literal character values in quotes: `"Record inserted."` (You can use either single apostrophes or double quotes.) **4GL** assumes any literal character value has the type CHAR(*n*) where *n* is the number of characters between the quotes.

You can also write literal values of types DATE, DATETIME, and INTERVAL; see the *INFORMIX-4GL Reference* for details on these data types.

Values from Variables

The next simplest kind of expression is the name of a simple variable or the name of one element of a structured variable. Suppose that the following names are declared and a value assigned to **pi**.

```
DEFINE
    j, prodNum INTEGER
    pi SMALLFLOAT
    LET pi = 3.1415926
```

Following these statements, you can use the name **pi** as an expression meaning `3.1415926` and having the type SMALLFLOAT. The names **j** and **prodNum** are expressions meaning, "the value that was last stored in this variable," and have a type of INTEGER.

References to structured variables are also expressions. The array access `custByProd[j,prod_num]` is an expression meaning, "the value that was last stored in the subscripted element of this array."

Values from Function Calls

A function can return one or more values. In fact, this is the most common reason to write a function: to calculate and return a value. (Functions are discussed in more detail under "Functions and Calls" on page 8-29.)

A function that returns a single value can be used as an expression.

You write a call to a function by writing the name of the function followed by its arguments enclosed in parentheses. See also "Data Type Conversion" on page 8-25.

The use of functions returning more than one value is covered under "Working with Multiple Values" on page 8-32.

Numeric Expressions

You can write expressions that calculate numeric values. You can use the arithmetic operators to combine numeric literals and the names of numeric constants, variables, and functions to calculate new values. The following statement from an example in the preceding chapter contains several numeric expressions:

```
    LET fahrDeg = (9*centDeg)/5 + 32
```

In this statement, 9, 5, and 32 are literals, and **centDeg** is the name of a variable. The expression `9*centDeg` tells **4GL** to produce a new value by multiplying two values, and `(9*centDeg)/5+32` tells it how to produce another new value by dividing and then adding.

As previously noted, **4GL** carries intermediate results of calculations on numbers with fractional components in its most precise data type, DECIMAL(32). This helps prevent many errors due to rounding and truncation, and reduces the chance that an intermediate result will overflow. However, it is still possible for a badly planned calculation to cause overflow, round-off, or truncation errors. As with any computer language, you should think through the precision requirements of any critical calculation.

Integer and numeric expressions are described in Chapter 3, *INFORMIX-4GL Reference*.

Relational and Boolean Expressions

In **4GL**, the result of comparing two values is a new integer value: 1 if the comparison is true and 0 if it is false. Suppose the following variables have been declared and initialized:

```
DEFINE maxRow, rowNum INTEGER
```

The expression `rowNum==maxRow` is a comparison; that is, a numeric expression with a value of either 1 or 0.

Note: *You can write an equality comparison using either a single equals sign or a double one. If you have used the C language, you may prefer to write "=="; if your experience has been with other languages you may prefer to use just one.*

Other relational operators include <> for "not-equals" and MATCHES for character pattern matching. Two especially important relational operators are IS NULL and IS NOT NULL. You can use these to test for the presence of a NULL value in a variable. (For more information on the use of NULL values, see "Null Values" on page 8-22 and Chapter 3, *INFORMIX-4GL Reference*.)

All these relational tests return 1 to mean "true" or 0 to mean "false." The names TRUE and FALSE are predefined **4GL** constants with those values.

You can combine numeric values with the Boolean operators AND, OR, and NOT. Normally you use them to combine the results of relational expressions, writing expressions such as `keepOn="Y" AND rowNum < maxRows`. That expression means:

- Take the value of the comparison `keepOn="Y"` (which is 1 or 0).
- Take the value of the comparison `rowNum<maxRows` (1 or 0).
- Combine those two values using AND to produce a new value.

Usually you write Boolean expressions as part of IF statements and other decision-making statements (for some examples, see the section "Decisions and Loops" on page 8-26). However, a comparison is simply a numeric expression. You can store its value in a variable, pass it as a function argument, or use it any other way that an expression can be used.

For more on relational and Boolean expressions, see Chapter 3 of the *INFORMIX-4GL Reference*.

Character Expressions

You can express a character value:

- With a literal.
- By naming a CHAR or VARCHAR variable.

You can produce a character value:

- With a call to a function that returns CHAR or VARCHAR.
- As a *substring* of a literal, a variable, or the returned value from a function returning a CHAR or VARCHAR value.

A substring is written as one or two numbers in square brackets. The first is the position of the first character to extract, and the second is the position of the last. Suppose this variable exists:

```
DEFINE delConfirm CHAR(11)
LET delConfirm = "Row deleted"
```

Now you can write `delConfirm[1,3]` as an expression with the value `Row` and the expression `delConfirm[5,8]` is the value `dele`. The expression `delConfirm[4]` or `delConfirm[4,4]` is a single space character.

It should be noted that the sub*string* expression uses the same notation as the subscript to an array. Here is an example of an array of character values:

```
DEFINE companies ARRAY[75] OF CHAR(15)
```

The expression `companies[61]` produces the character value from the 61st element of the array. The expression `companies[1,7]` would cause an error at compile time because the array **companies** does not have two dimensions. However, the expression `companies[61][1,7]` accesses the 61st element and then extracts the first through the seventh letters of that value.

Null Values

For every data type, the Informix database engines define a NULL value. A value of NULL in a database column means *do not know*, or *not applicable*, or *unknown*. Since null values can be read from the database into program variables, **4GL** also supports a NULL value for every data type. The keyword NULL stands for this *unknown* value. A variable of any type can contain NULL and a function can return a variable with a value of NULL.

Null Values in Arithmetic

If you do arithmetic that combines a number with NULL, the result is NULL. Adding 1 to *unknown* must result in *unknown*. Thus a single NULL value in an arithmetic expression will usually make the entire expression have the value NULL.

Null Values in Comparisons

If you compare a NULL value to anything else, the result is NULL. Is "A" equal to *unknown*? The only answer can be *unknown*. Thus every comparison expression has not two but three possible results: TRUE, FALSE, and *unknown*, or NULL.

It is worth noting that the decision-making statements such as IF and WHILE do not distinguish this third result. They treat NULL the same as FALSE. This can cause problems when you test values that may be NULL. That is why the relational tests IS NULL and IS NOT NULL are provided; they allows you to detect NULL values and respond according to the requirements of your application program.

Null Values in Boolean Expressions

The Boolean operators AND and OR give special treatment to NULL. In the case of OR, when one of its arguments is TRUE, its result is TRUE no matter what the other argument may be. But if one of its arguments is FALSE and the other is NULL, OR must return NULL; it does not know whether its result should be TRUE or FALSE.

Assignment and Data Conversion

In creating a useful **4GL** program, first you write expressions to describe values; then, you do something with the values. Sometimes you display them or pass them as arguments to functions. Most often, you *assign* a value for a variable; that is, you tell **4GL** to store the value in the memory reserved to the variable.

```
DEFINE fahrDeg, centDeg DECIMAL(4,2)
LET centDeg = 10.28
LET fahrDeg = (9*centDeg)/5 + 32
```

The expression is `(9*centDeg)/5+32`. The LET statement causes **4GL** to calculate the value of the expression and to store that value in the memory reserved for the variable **fahrDeg**.

There are several ways to assign a value to a variable:

- Use the LET statement. This is the most common kind of assignment.
- Use the CALL…RETURNING statement to call a function and store the value it returns.

```
CALL tempConvert(32) RETURNING fahrDeg
```

- Use the INTO clause of an SQL statement to get a value from the database and assign it to a variable.

```
SELECT COUNT(*) INTO maxRow FROM stock
```

  Other SQL statements that support INTO include SELECT, FETCH, and FOREACH.

- Use the INITIALIZE statement to set a variable, or all members of a record, to NULL or to other values.

```
INITIALIZE custRow.* TO NULL
```

- Use PROMPT to accept values entered by the user from the keyboard.

```
PROMPT "Enter temp to convert: " FOR centDeg
```

- Use INPUT or INPUT ARRAY to get values from fields of a form and put them in variables.

```
INPUT custRec.* FROM customer.*
```

- Use CONSTRUCT to get a query by example expression from a form and put it in a variable.

```
CONSTRUCT BY NAME whereClause ON customer.*
```

Data Type Conversion

All of the preceding methods of assignment perform the same action: they store values in variables. Each performs automatic *data conversion* when is necessary and possible. Data conversion is necessary when the data type of the value is different from the data type of the variable that receives it.

As previously noted, **4GL** has liberal rules for data conversion. It will attempt to convert a value of almost any type to match the type of the receiving variable. For a table summarizing the rules and showing data type incompatibilities, see Chapter 3 of *INFORMIX-4GL Reference* for a table of compatible **4GL** data types.

```
DEFINE num DECIMAL(8,6)
DEFINE chr CHAR(8)
LET num = 2.18781
LET chr = num
```

The second assignment statement asks **4GL** to initialize **chr**, a character variable, from the value of **num**, a numeric variable. In other words, this statement asks **4GL** to convert the value in **num** to character. It does that using the same rules it would use when displaying the number, in this case producing the string `2.187810` (with all six declared decimal places filled in).

```
LET num = chr[1,3]
```

Given the initialization of **chr** to the string `2.187810`, the expression `chr[1,3]` returns the characters `2.1`. Since the receiving variable has type `DECIMAL(8,6)`, **4GL** converts the characters into a number `2.100000` and assign that to **num**.

Conversion Errors

Some conversions cannot be done. When **4GL** can recognize at compile time that a particular conversion is illegal, it returns a compiler error.

Two data types that **4GL** never attempts to convert are `BYTE` and `TEXT`. The reason is the same in each case; **4GL** does not know enough about the internal structure of values of these types to convert them.

Some conversions may only prove to be impossible at execution time. Then the error will be detected while the program is running. For example, the following program tries to assign a CHAR value to a SMALLINT variable:

```
DATABASE stores2
DEFINE a, b SMALLINT, c,d CHAR(10)

MAIN
LET c="apple"
DISPLAY "This is c ", c AT 3,3
SLEEP 2

LET a=c
DISPLAY "This is a ", a AT 5,5
SLEEP 4

END MAIN
```

For more information, see "Run-Time Errors" on page 12-4.

Decisions and Loops

The statements you use to control the sequence of execution are similar to those in other languages you may have used. You will find details in the *INFORMIX-4GL Reference* article for each statement.

IF…THEN…ELSE Tests for Boolean (yes / no) conditions. You write the test as a conditional statement, usually a relational comparison or a Boolean combination of relational comparison. If the value of the expression is 1 (TRUE), the THEN statements are executed. When the expression evaluates to 0 (FALSE) or is NULL, the ELSE statements are executed.

```
    IF promptAnswer MATCHES "[yY]" THEN
        DELETE WHERE CURRENT OF custCursor
    ELSE
        DISPLAY "Row not deleted at your request"
    END IF
```

CASE(*expr*) Implements multiple branches. This statement has two forms. The first is a simple form that compares one expression for equality against a list of possible values.

```
    CASE (WEEKDAY(ship_date))
    WHEN 0  -- Sunday
        DISPLAY "Will ship by noon Monday"
    WHEN 5  -- Friday
        DISPLAY "Will ship by noon Saturday"
    WHEN 6  -- Saturday already
        DISPLAY "Will ship by noon Monday"
    OTHERWISE
        IF DATETIME (12) HOUR TO HOUR <
                EXTEND(CURRENT,HOUR TO HOUR) THEN
            DISPLAY "Will ship by 5 today"
        ELSE -- past noon
            DISPLAY "Will ship by noon tomorrow"
        END IF
    END CASE
```

CASE The second form of CASE is effectively a list of *else-if* tests. No expression follows the keyword CASE, but a complete Boolean expression (instead of a comparison value) follows each WHEN keyword.

```
MAIN
DEFINE promptAnswer CHAR(10)
PROMPT "Delete current row?  " FOR promptAnswer
CASE
    WHEN promptAnswer MATCHES "[Yy]"
        DISPLAY "Row will be deleted." AT 2,2
    WHEN promptAnswer MATCHES "[Nn]"
        DISPLAY "Row not deleted." AT 2,2
    WHEN promptAnswer MATCHES ("Maybe")
        DISPLAY "Please make a decision." AT 2,2
    OTHERWISE
        DISPLAY "Please read the instructions again." AT 2,2
END CASE
SLEEP 5
END MAIN
```

WHILE Provides for generalized looping. You can use the EXIT statement to break out of a loop early.

```
LET j=1
WHILE manyObj[j] IS NOT NULL
    LET j = j + 1
    IF j > maxArraySize THEN -- off the end of the array
        LET j = maxArraySize
        EXIT WHILE
    END IF
END WHILE
DISPLAY "Array contains ",j," elements."
```

FOR Provides counting loops.

```
FOR j = 1 TO maxArraySize
    IF manyObj[j] IS NOT NULL THEN
        LET j = j-1
        EXIT FOR
    END IF
END FOR
DISPLAY "Array contains ",j," elements."
```

An additional loop, the FOREACH loop, is discussed under "Row-by-Row SQL" on page 9-5.

Decisions Based on Null

If a Boolean comparison evaluates to NULL (see "Null Values" on page 8-22), it will have the same effect as FALSE:

- IF NULL… always executes the ELSE statements (if any).
- CASE (NULL)… always executes the OTHERWISE statements (if any).
- WHILE NULL… does not execute its loop statements at all.

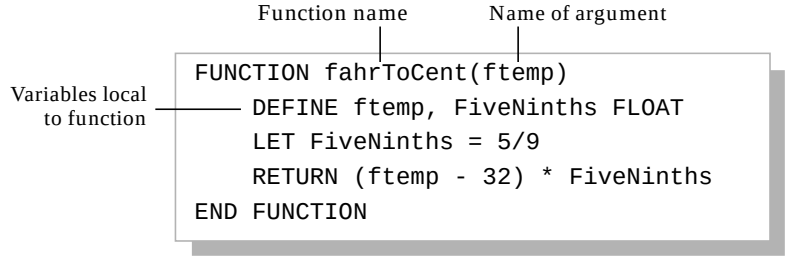
Using a NULL value as either the starting or the ending number in a FOR loop results in an endless loop. The FOR loop ends when the control variable equals the upper limit, but a NULL value cannot equal anything; hence the loop never ends.

Functions and Calls

A function is a named block of executable code. The function is your primary tool for achieving a readable, modular program.

Function Definition

You *define* a function when you specify the executable statements it contains. Here is a definition for a simple function:



This example shows the important parts of a function definition. It contains:

- A FUNCTION statement that defines:
  - The name of the function (**fahrToCent** in the example).
  - How many arguments it takes (just one in the example).
- A function *program block*, statements between the FUNCTION and END FUNCTION.
  - DEFINE statements must appear first, before other kinds of statements.
  - Executable statements do the work of the function. In the example there is only one, a RETURN statement.

Variables declared in the program block are local to the function. The variable named **FiveNinths** is local to this function; it is not available outside the function, although other **FiveNinths** variables can be declared at the module level, locally in other functions, or globally.

When the **4GL** compiler processes a function definition, it generates the executable code of the function.

Once defined, a function is available to any **4GL** module in your program.

Invoking Functions

You cause a function to be executed by *calling* it. There are two ways to call a function:

- In an expression
- Through the CALL statement

When a function returns a single value, you can call it as part of an expression. The **fahrToCent( )** function described previously returns a single value, so it can be called in an expression that expects the data type of the returned value.

```
    LET tmp_range = fahrToCent(maxTemp) - fahrToCent(minTemp)
```

This statement contains two calls to the function **fahrToCent( )**. The statement subtracts one of these values from the other and assigns the result to a variable of type FLOAT named **tmp_range**.

When a function returns no values or multiple values, you must use the CALL statement. Functions that return one value can be called in this way also.

```
    CALL mergeFiles()
    CALL fahrToCent(currTemp) RETURNING cTemp
```

The very useful ability to return more than one value from a function is considered further under "Working with Multiple Values" on page 8-32.

Arguments and Local Variables

The arguments you provide when a function is called are, in effect, local variables of the function. That is, these names (the name *ftemp* in the **fahrToCent( )** function, for example) represent values that are passed to the function when it is called. They are local to the function.

The following things happen when a function is called:

- The local variables of the function are allocated in memory, including the variables that will represent the arguments. In the function **fahrToCent( )**, local variables are **fiveNinths** and **ftemp**, its argument.

- Each argument expression in the function call is evaluated. In the following call, the expression `targetTemp + 20` is evaluated.

```
LET limitTemp = fahrToCent(targetTemp + 20)
```

- Each argument value is assigned to its argument variable, as described earlier ("Assignment and Data Conversion" on page 8-23). When an argument value has a different type from the argument variable, **4GL** attempts to convert it, as it would in any assignment.
- The statements of the function are executed.
- The local variables, including the argument variables, are discarded and the memory reclaimed.

The key point here is that the expressions you write in the call to a function are, in effect, assignments to local variables of the function. Knowing this, you can answer some common questions:

- Does a value passed to a function require a certain data type?

  No, because the value is assigned to the argument variable, and **4GL** *will attempt to convert it to the specific type.*

- Can a function assign new values to its arguments?

  Yes, because they are simply local variables.

- Does this change the contents of variables named in the call to the function?

  No, because the function's argument variables are local to it.

This method of passing arguments to functions is known as *call by value*. An alternative technique, *call by reference*, is used in some other programming languages, but generally not by **4GL**. The only call by references in **4GL** are references to BYTE and TEXT data types. These are called by reference because it is not practical to pass blobs by value.

The use of call by value has an effect on performance. Each argument value is copied into the function's variable. When the arguments are bulky character strings, the time such copying takes can be significant. A common way of avoiding this time penalty is to use global variables.

Working with Multiple Values

4GL lets you work with record variables in a very consistent and flexible way. The basic rules for records are:

- The name of a record followed by a dot and an asterisk, *record.**, also means a list of all the members of the record.
- You can select a range of members using *record.first* THRU *last* where *first* and *last* are names of members of *record*.

These examples illustrate the use of these rules:

```
DEFINE
    rSSS1, rSSS2 RECORD s1, s2, s3 SMALLINT END RECORD
    rFFC RECORD f1, f2 FLOAT , c3 CHAR(7) END RECORD

FUNCTION takes3(a,b,c)
DEFINE a,b,c SMALLINT
...
END FUNCTION
```

These statements define three record variables and declare a function that takes three arguments. The function **takes3()** will be used in examples in subsequent sections.

Assigning One Record to Another

To assign a value to a single member of a record, you use LET.

```
LET rSSS1.s1 = 101
LET rSSS1.s2 = rSSS1.s1 + 1
LET rSSS1.s3 = 103
```

You can assign one record to another using LET when they have the same number of members.

```
LET rSSS2.* = rSSS1.*
```

This statement assigns the three members of **rSSS1** to the corresponding members of **rSSS2**.

In other words, **4GL** assigns members one at a time, with data conversion as required. The members must all have simple data types, and the data types must be the same, or else data conversion must be possible.

You can use THRU notation to assign a range of members.

```
LET rFFC.f1 THRU rFFFC.f2 = rSSS1.s2 THRU rFFFC.s3
```

Note: *In general, THRU is allowed whereever an expression list is allowed, such as in a CALL statement. However, THRU in the LET statement is allowed in a special case only:*

LET charvar = rec.a THRU rec.b

Passing Records to Functions

The name of a record is a list of values; and a function takes a list of arguments. Thus you can use a record as a list of arguments.

```
CALL takes3(rFFC.*)
```

The previous statement is equivalent to listing the members:

```
CALL takes3(rFFC.f1,rFFC.f2,rFFC.c3)
```

When calling a function, you can also mix record members and single expressions as arguments.

```
CALL takes3(17, rSSS1.f2 THRU rSSS1.f3)
```

Returning Records from Functions

A function can return more than one value. You can make this happen by writing a RETURN statement in the FUNCTION definition containing a list of expressions. An example follows.

The function **agedBalances( )** returns the amounts that are owed by a specified customer as three numbers: amounts owed for 30 days or less, 31-60 days, and more than 60 days.

```
DATABASE stores2
FUNCTION agedBalances(cn)
DEFINE cn LIKE customer.customer_num ,
         bal30, bal60, bal90 DEC(8,2) ,
         ordDate LIKE orders.order_date ,
         ordAmt DEC(8,2)
    LET bal30 = 0.00
    LET bal60 = 0
    LET bal90 = 0

    DECLARE balCurs CURSOR FOR
        SELECT order_date, SUM(items.total_price)
        FROM orders, items
        WHERE orders.customer_num = cn
        AND orders.order_num = items.order_num
        GROUP BY order_date

    FOREACH balCurs INTO ordDate, ordAmt
        IF ordDate <= TODAY - 90 THEN
            LET bal90 = bal90 + ordAmt
        ELSE IF ordDate <= TODAY - 60 THEN
            LET bal60 = bal60 + ordAmt
        ELSE
            LET bal30 = bal30 + ordAmt
        END IF
        END FOREACH
    RETURN bal30, bal60, bal90
END FUNCTION
```

The RETURN statement must match in number of values in the calling function.

A function like this one can be used in several ways. It can be used in a CALL…RETURNING statement. You list variables to receive the values.

```
    DEFINE balShort, balMed, balLong MONEY(10)
    ...
    CALL agedBalances(custNumber)
        RETURNING balShort, balMed, balLong
```

If you have a record with appropriate members of the appropriate number of data types, you can refer to it in the RETURNING clause of the CALL statement.

```
    DEFINE balRec RECORD b1, b2, b3 MONEY(10) END RECORD
...
    CALL agedBalances(custNumber) RETURNING balRec.*
```

Chapter

Using Database Cursors

9

Overview

In many ways the SQL language can be considered a subset of the **4GL** language because you can *embed* many SQL statements in a **4GL** program.

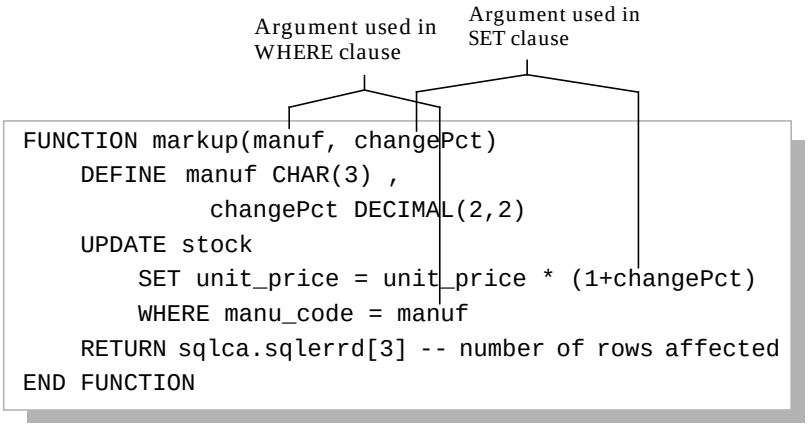
The SQL Language

SQL can be used both procedurally and non-procedurally, depending on the needs of your application. Similarly, SQL statements can be static—that is, created at compile time—or dynamic. Dynamic statements are composed during run time, based in whole or in part on information supplied or selected by the user of the application.

For additional information about the use of SQL in general and embedded in a **4GL** program in particular, see the *Informix Guide to SQL: Tutorial*, and the *Informix Guide to SQL: Reference*. If you want to use 5.0 or 6.0 SQL features, see the *Informix Guide to SQL: Syntax*. However, to include 6.0 SQL statements in a a **4GL** program, you must prepare the statement (by using the PREPARE statement). For information on preparing SQL statements, see Chapter 3 of the *INFORMIX-4GL Reference*.

Nonprocedural SQL

Here is an example of SQL use shown earlier in this book. It defines a function named **markup**( ) whose purpose is to alter the prices of stock received from a specified manufacturer.




```
FUNCTION markup(manuf, changePct)
    DEFINE manuf CHAR(3) ,
             changePct DECIMAL(2,2)
    UPDATE stock
        SET unit_price = unit_price * (1+changePct)
        WHERE manu_code = manuf
    RETURN sqlca.sqlerrd[3] -- number of rows affected
END FUNCTION
```

The function takes two arguments. The first, **manuf**, is the code for the supplier whose prices are to be changed. The second, **changePct**, is the fraction by which prices should be changed.

Here is an example call to **markup( )**:

```
LET rowCount = markup("ANZ",0.05)
DISPLAY rowCount, " stock items changed."
```

The SQL statement UPDATE in the definition of the **markup( )** function causes a change in the unit prices of certain stock items in the database. The function argument values are used in this UPDATE statement, one in the SET clause and one in the WHERE clause.

This function is an example of the nonprocedural use of SQL. The UPDATE statement will examine many rows of the **stock** table. It may update all, some, or none of them. The **4GL** program does not loop, updating rows one at a time; instead it specifies the set of rows using a WHERE clause and leaves the sequence of events to the database engine.

Nonprocedural SELECT

All 4.1-level SQL statements except the SELECT statement can be used this way: by writing them in the body of a function. SELECT also can be used this way as long as only a single row is returned. The following function returns the count of unpaid orders for a single customer, given the customer name:

```
FUNCTION unpaidCount(cust)
    DEFINE  cust LIKE customer.company ,
            theAnswer INTEGER
    SELECT COUNT(*) INTO theAnswer
        FROM customer, orders
        WHERE customer.company = cust
        AND customer.customer_num = orders.customer_num
        AND orders.paid_date IS NULL
    RETURN theAnswer
END FUNCTION
```

Since the SELECT statement returns only an aggregate result (a count), it can return only a single value. The argument variable **cust** is used in the WHERE clause. The result of the SELECT is assigned to the local variable **theAnswer** by the INTO clause.

Some SQL statements do not allow program variables in all contexts. You can refer to the syntax diagrams in *Informix Guide to SQL: Syntax*.

Note: *To include any SQL statements introduced after version 4.1 of the servers in a* **4GL** *program, you must prepare the statement. You prepare a statement by using the PREPARE statement. For a list of SQL statements that must be prepared and for information on preparing SQL statements, see Chapter 3 of the INFORMIX-4GL Reference.*

Row-by-Row SQL

When a SELECT statement can return more than one row of data, you must write procedural logic to deal with each row as it is retrieved. You do this in four or five steps:

- If you wish to generate your SQL statement dynamically—that is, using the **4GL** CONSTRUCT statement to generate dynamic search criteria—place your statement text in a CHAR variable and use the SQL PREPARE

statement. (See "Dynamic SQL" on page 9-10 and the *Informix Guide to SQL: Reference*.)

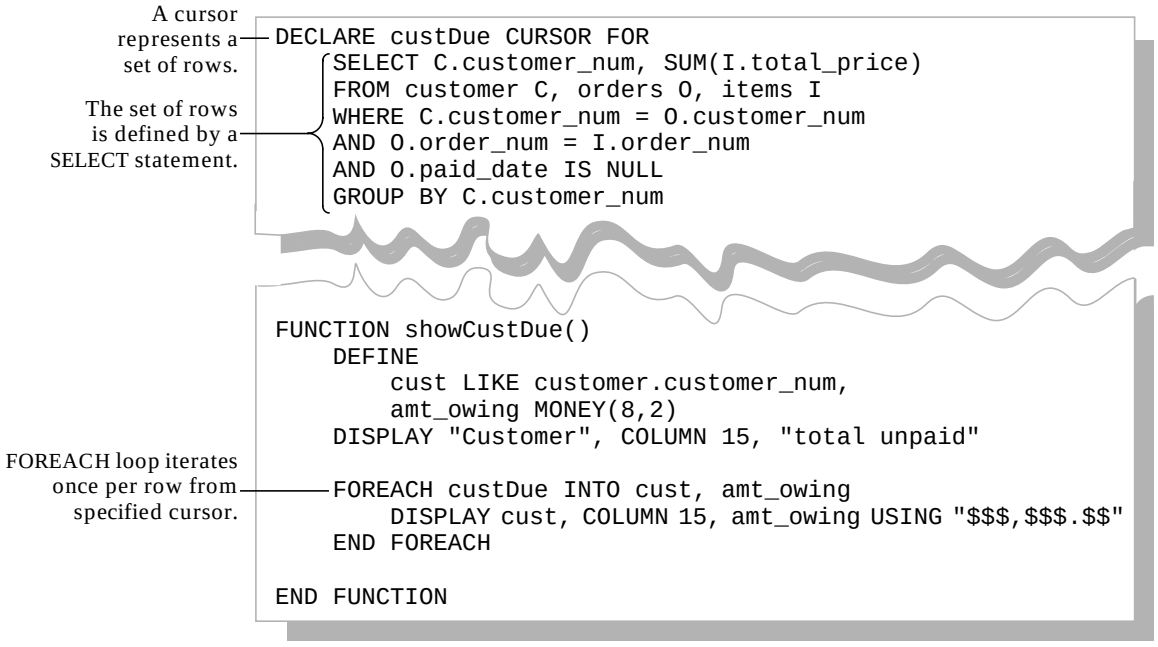
- You declare a database *cursor*, which is a name that stands for a selection of rows.

  You specify the rows using SELECT statement. While you often specify a selection from a single table, you are free to specify rows formed by unions and / or joins over many tables, and including calculated values, literal values, aggregates, and counts.

- You open the cursor, causing the database engine to retrieve the first of the specified set of rows.
- You retrieve rows one at a time through the cursor using the FETCH statement and process each one as it is produced.
- You close the cursor, releasing the set of rows.

Alternatively, you can use FOREACH statement to automatically open the cursor, FETCH one row for each traversal of the FOREACH loop, and then close the cursor after you have processed the last row of the selection set.

The following example contains a SELECT statement that retrieves one row for each customer that has an unpaid order in the demonstration database. The selected data consists of the customer number and the total value of that customer's unpaid orders. The DECLARE statement must be executed before FOREACH in **showCustDue( )**.



Unless used in conjunction with a PREPARE statement, the SELECT statement is written within a DECLARE statement, which creates a database cursor. The cursor, when opened, represents the set of all selected rows. (For more on database cursors and active sets, see the *Informix Guide to SQL: Tutorial*.)

The FOREACH statement in **4GL** has three effects:

- It opens the database cursor.
- For every row in the selected set, FOREACH:
  - Fetches the column values for that row. (In the example, it assigns the fetched values to local variables **cust** and **amt_owing**.)
  - Executes the statements in the body of the loop (a single DISPLAY in the example above).
- Closes the cursor.

This is a common pattern for many programs: open a cursor, fetch the rows and process each row, and close the cursor. The step "process each row," of course, can be very elaborate, especially when you "process" a row by displaying it in a screen form for the user to read or change.

Updating the Cursor's Current Row

When you have fetched a row of a single table (not a row produced by joining tables) through a cursor, you can delete or update that particular row. You do this by:

- Making the cursor an *update cursor*, which locks the selected row
- Then indicating in your UPDATE statement that you wish to update the current cursor row

To make the cursor an update cursor, add the keywords FOR UPDATE to the cursor declaration; you can also limit the update to certain columns by specifying those column names in the FOR UPDATE clause. When using the cursor, you can update or delete the current row by writing an UPDATE or DELETE statement as usual and adding the clause WHERE CURRENT OF *cursor*, supplying the name of the cursor from which you fetched the row.

The following function uses a cursor to scan the **orders** table and deletes any row for which the paid-date is at least a month old. (Note that the same task could more easily be accomplished by a nonprocedural UPDATE.)

```
DECLARE oldOrder CURSOR FOR
    SELECT order_num, paid_date FROM orders
    — — no WHERE clause, all rows scanned
    FOR UPDATE
FOREACH oldOrder INTO o_num, p_date
    IF 30 < (TODAY - o_num) THEN
        DELETE FROM orders WHERE CURRENT OF oldOrder
    END IF
END FOREACH
```

The clause FOR UPDATE tells the database engine that you may update or delete fetched rows (it is not required in an ANSI-compliant database).

Updating Through a Primary Key

Often you will find reasons why you cannot or should not update the current row through the same cursor. (For example, the cursor produces rows based on a join of multiple tables.) When this is the case, you can use the nonprocedural UPDATE or DELETE statement instead.

Be sure that the cursor produces a row that contains the primary key of the row to be updated, so that you can isolate the exact row you want to modify. When you fetch a row, you fetch the values of its primary key into program variables.

If you decide to change the row, you execute the UPDATE or DELETE statement containing a WHERE clause that selects the specific row based on its primary key values.

Updating with a Second Cursor

The following situation arises quite often. You want to select a set of rows using a cursor. You will display each row on the screen and wait for the user to react. The user may then tell you to delete or update the displayed row.

This is not a problem when your program is the only one using the database; you can use UPDATE …WHERE CURRENT OF or you can update using the primary key, whichever is appropriate.

However, the situation does become difficult when multiple users may be working in the same table at the same time, using multiple copies of your program or using different programs. You do not want to lock the row while your user examines it; your user might answer the telephone or go to lunch, blocking other users out. Hence you do not want to select rows using an update cursor, which locks rows.

The answer is to use two cursors. The first, primary cursor selects the rows of interest. You include in each row the primary key column(s). The second cursor selects only one row based on its primary key, and is declared FOR UPDATE. When the user chooses to update the current row, proceed as follows:

1. Open the second cursor.
2. Fetch the one matching row into a temporary record. If the row with this ROWID value cannot be found, you know that another user must have deleted it while your user was looking at the screen display.
3. Compare the second set of column values to the ones you displayed to the user. If any important ones have changed, you know that some other user has altered this row while your user was looking at the display. Notify your user and do not proceed.
4. Update the row through the second cursor using WHERE CURRENT OF.
5. Close the second cursor.

You can find examples of this kind of programming in *INFORMIX-4GL by Example*.

Dynamic SQL

In the preceding examples, the SQL statements are *static*. That is, they were written into the program source, and hence are static in that their clauses are fixed at the time the source module is compiled. Only the values supplied from program variables can be changed at execution time.

There are many times when you need to generate the contents of the SQL statement itself while the program is running. For instance, you probably want users of your program to be able to retrieve records based upon queries they devise during the day-to-day operation of their business. In other words, in *real time*. When you do this, you are using *dynamic* SQL.

The following function uses dynamic SQL. It assembles the text of a GRANT statement and executes it. It takes three arguments:

- The name of the user to receive the privilege.
- The name of a table on which the privilege is to be granted.
- The name of a table-level privilege to be granted (for example, INSERT).

```
FUNCTION tableGrant( whom , tab , priv )
    DEFINE whom , tab , priv CHAR(20), granTextCHAR(100)
    LET granText = "GRANT " , priv , " ON " , tab ,
                   " TO " , whom
    PREPARE granite FROM granText
    EXECUTE granite
END FUNCTION
```

This function does nothing about handling the many possible errors that could arise in preparing and executing this statement. In "Using WHENEVER in a Program" on page 12-12, you can find a version of the same program that does handle errors.

Chapter

10

Creating Reports

Overview

4GL reports are introduced in "Creating 4GL Reports" on page 6-4. As noted there, a report program has two parts:

- A *report driver* that produces rows of data
- A *report formatter* that sorts the rows (if necessary), creates subtotals and other summary information, and formats the rows for output

When you design a report program you can design these two parts independently. A report driver can produce rows for any number of reports.

The **4GL** statements you use for reports are covered in detail in Chapter 6 of the *INFORMIX-4GL Reference*. And you can find several examples of programs that produce reports distributed with **4GL**.

Designing the Report Driver

The report driver executes the following steps:

- Initialize the report using the START REPORT statement.

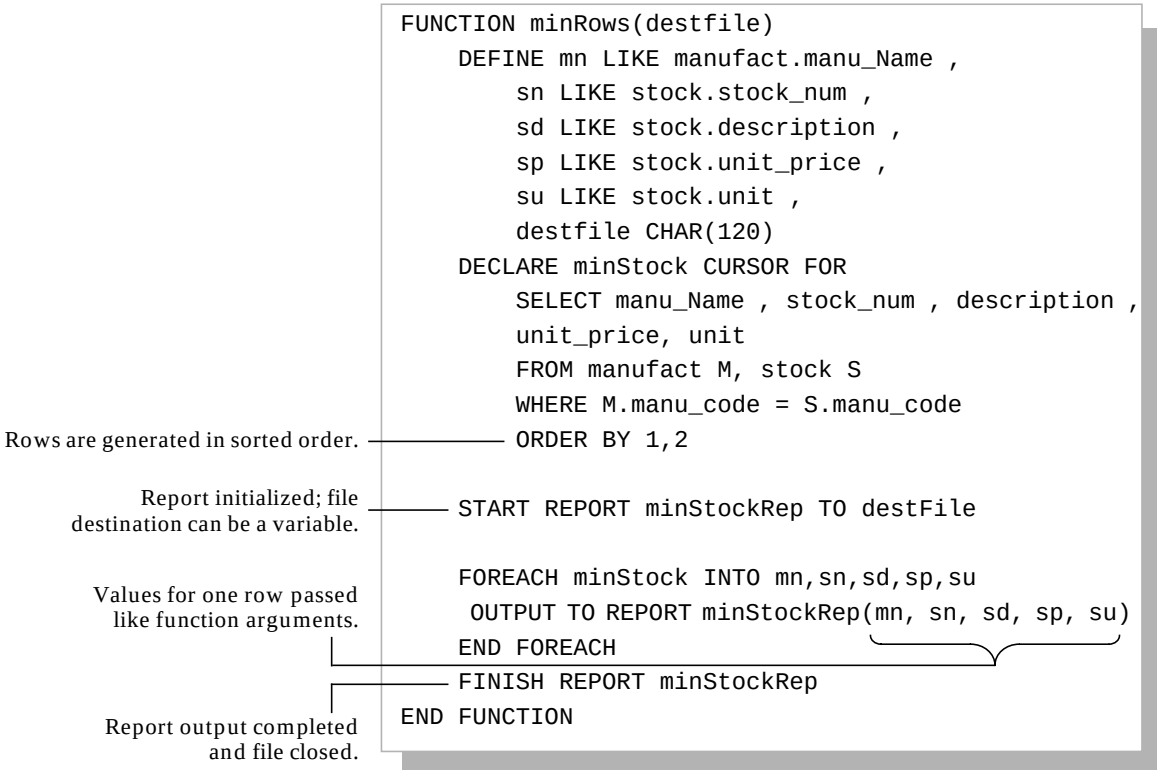
  This statement initializes the report formatter. It can also specify the destination of the report such as the screen, the printer, a file, or another program.
- Generate rows of data, sending each row using OUTPUT TO REPORT.

  This statement, which is similar to a function call, passes one row of data to the report. Although called a "row," each group of data values need not come from a row of a database table; the values can come from any source, including calculation made by your program. It is equally valid to look at a *row* as an *input record*.
- Conclude row processing.
- Terminate the report using FINISH REPORT.

Totals and other aggregates are calculated, the report is output, and the output file is closed.

An Example Report Driver

Row production can be a natural part of a **4GL** application. Here is a brief example of row-producing code (the report itself appears on page 10-7):



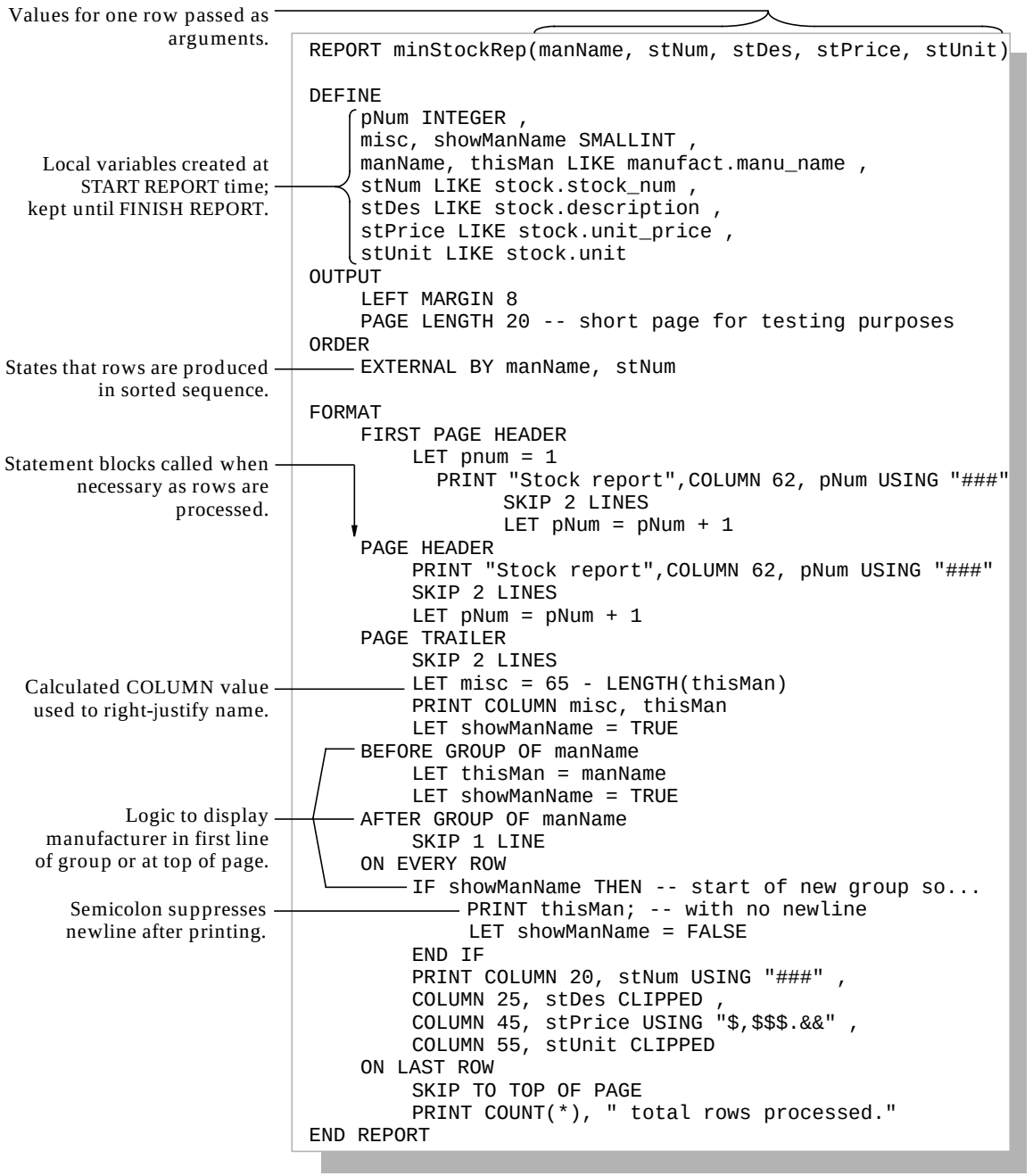
This function takes a filename (it may be a complete pathname) as its argument, and produces a report with that destination.

The values in each row describe one row of merchandise from the stores demonstration database. The values are produced by a database cursor defined on a join of the **stock** and **manufact** tables. They are produced in sorted order using the row ordering capability of the database engine.

The report formatter is named **minStockRep( )**. That name appears in the START, OUTPUT, and FINISH statements.

Designing the Report Formatter

Although a report has the general form of a function, its contents are quite different. The body of a function contains one block of statements, while the body of a report contains several independent statement blocks that are executed as needed. Here is the **minStockRep( )** that completes the preceding example. This code is examined in detail in the topics that follow.




```
REPORT minStockRep(manName, stNum, stDes, stPrice, stUnit)

DEFINE
    pNum INTEGER ,
    misc, showManName SMALLINT ,
    manName, thisMan LIKE manufact.manu_name ,
    stNum LIKE stock.stock_num ,
    stDes LIKE stock.description ,
    stPrice LIKE stock.unit_price ,
    stUnit LIKE stock.unit
OUTPUT
    LEFT MARGIN 8
    PAGE LENGTH 20 -- short page for testing purposes
ORDER
    EXTERNAL BY manName, stNum

FORMAT
    FIRST PAGE HEADER
        LET pnum = 1
          PRINT "Stock report",COLUMN 62, pNum USING "###"
                SKIP 2 LINES
                LET pNum = pNum + 1
    PAGE HEADER
        PRINT "Stock report",COLUMN 62, pNum USING "###"
        SKIP 2 LINES
        LET pNum = pNum + 1
    PAGE TRAILER
        SKIP 2 LINES
        LET misc = 65 - LENGTH(thisMan)
        PRINT COLUMN misc, thisMan
        LET showManName = TRUE
    BEFORE GROUP OF manName
        LET thisMan = manName
        LET showManName = TRUE
    AFTER GROUP OF manName
        SKIP 1 LINE
    ON EVERY ROW
        IF showManName THEN -- start of new group so...
            PRINT thisMan; -- with no newline
                LET showManName = FALSE
        END IF
        PRINT COLUMN 20, stNum USING "###" ,
        COLUMN 25, stDes CLIPPED ,
        COLUMN 45, stPrice USING "$,$$$.&&" ,
        COLUMN 55, stUnit CLIPPED
    ON LAST ROW
        SKIP TO TOP OF PAGE
        PRINT COUNT(*), " total rows processed."
END REPORT
```

Figure 10-1 Typical 4GL report specification

The following is an excerpt from the output of this report. The page length was set to 20 for testing; it would normally be longer.

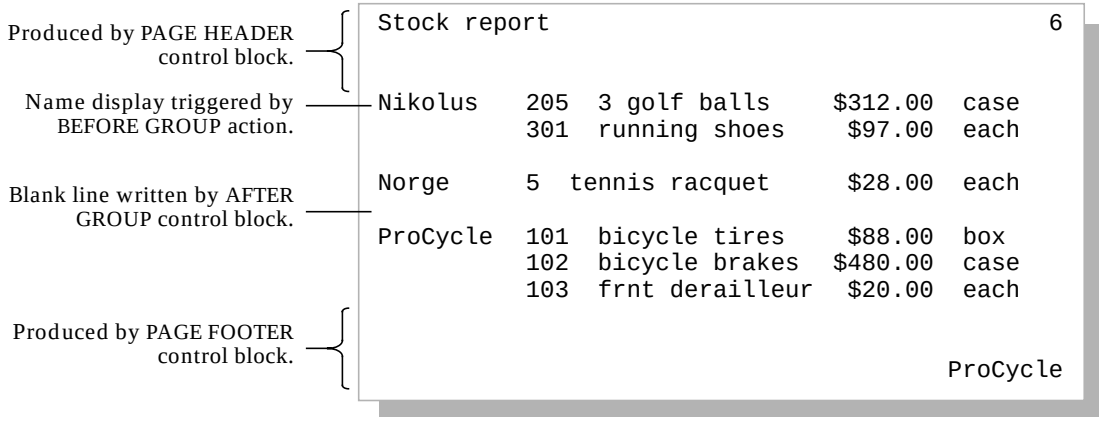


Figure 10-2 A sample 4GL report

The REPORT Statement

All reports begin with a REPORT statement. This statement is similar to a FUNCTION statement: it states a name, which becomes the name of the report, and a list of arguments. Here is the beginning of the **minStockRep( )** report definition:

```
REPORT minStockRep(manName, stNum, stDes, stPrice, stUnit)

DEFINE
    pNum INTEGER ,
    misc, showManName SMALLINT ,
    manName, thisMan LIKE manufact.manu_name ,
    stNum LIKE stock.num ,
    stDes LIKE stock.description ,
    stPrice LIKE stock.unit_price ,
    stUnit LIKE stock.unit
```

The report name is used to identify this report in the START REPORT, OUTPUT TO REPORT, and FINISH REPORT statements within the report driver.

The arguments to the report are the values that make one row of report data. This report takes five arguments. For the purposes of this report, one set of these values makes a row. Within the body of the report you can define and then refer to these names to find values of the current report row.

When the report driver code executes OUTPUT TO REPORT, it sends another set of values (that is, another row), to the report for processing. Here is such a statement from the example on page 10-4:

```
FOREACH minStock INTO mn, sn, sd, sp, su
    OUTPUT TO REPORT minStockRep( mn, sn, sd, sp, su )
END FOREACH
```

The Report Declaration Section

A report declares local variables, much like a function. Their definition must come immediately after the REPORT statement.

The **minStockRep( )** report defines several local variables using the LIKE keyword. Just as you can define the local variables of a function using LIKE, you can use LIKE to get a data type from the database for a report variable.

The local variables of a report are created and initialized when the START REPORT statement is executed. They remain in existence until the report is ended by FINISH REPORT. They are not reinitialized each time OUTPUT TO REPORT is executed. (This is one way a report differs from a function. The local variables of a function are created anew each time it is called.)

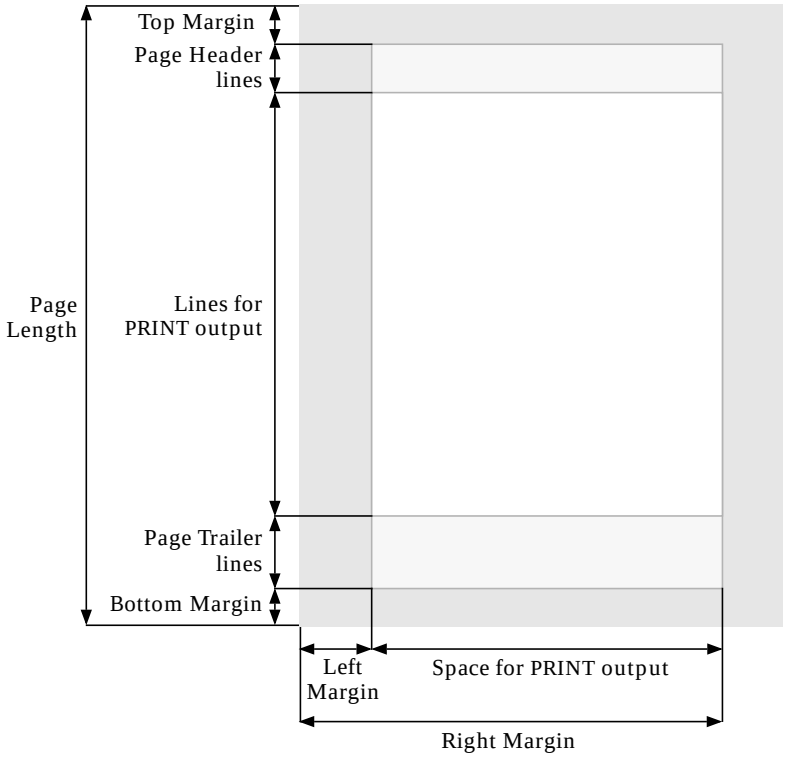
The OUTPUT Section

The OUTPUT section of a report is executed and takes effect at START REPORT time. The values it sets cannot be changed until the report is started again.

This section contains statements that set the basic format of the report. Five of them establish page layout:

LEFT MARGIN	Number of spaces inserted to the left of every print line.
RIGHT MARGIN	Total number of printed characters in any line, including left margin spaces. (This statement is ignored unless the FORMAT EVERY ROW default report formatting option is used.)
TOP MARGIN	Number of blank lines to print above the page header.

BOTTOM MARGIN	Number of blank lines to print after the page trailer (also known as a *footer*).
PAGE LENGTH	Total number of lines per page, including margins and page header and trailer sections.



The TOP OF PAGE specification is used in the OUTPUT section to specify a character value that **4GL** will use to cause a page-eject. If you omit it, **4GL** gets to a new page by printing empty lines.

The REPORT TO specification allows you to specify a report destination. (If a destination is given in the START REPORT statement, it takes precedence.) Each of these specifications takes a character expression.

The ORDER BY Section

The ORDER BY section of a report specifies whether the rows of data are to be sorted, and if so, whether or not they will be produced in sorted sequence. You must decide among three cases:

1. The order of the rows is not important; that is, the report is simply a list of rows in the order they happen to be generated. To choose this, you omit the ORDER BY section entirely.

   When ORDER BY is not used, you cannot process rows in groups or take aggregate values over them.

2. The rows should be sorted, so they will be processed in a specified order. To say this, you specify ORDER EXTERNAL and list the field on which the sort take place.

   If the report driver code fails to generate the rows in their proper sequence, the report output will be incorrect.

3. The rows need to be sorted, but the row-producing code does not produce them in the correct order. To say this, you write ORDER BY and specify the fields to sort on. You do not use EXTERNAL in this case.

When rows are to be sorted, it is best if the report driver code can produce them in correct order. When the rows come from the database, you can use the ORDER BY clause of the SELECT statement. The database engine has the most efficient ways of producing sorted rows.

When it is necessary that the report itself order the rows, but your report driver cannot produce the necessary order, **4GL** uses *two-pass report logic*. This is discussed in the section titled "One-Pass and Two-Pass Reports" on page 10-11.

Sort Keys

You can use the ORDER BY statement to specify the *sort keys* of the report. Here is the **minStockRep( )** ORDER statement. It specifies that rows should be sorted, and that they are produced by the report driver in sorted order.

```
ORDER
    EXTERNAL BY manName,stNum
```

The priority of the sort keys decreases from first to last; that is, the first one named is the major sort key. In the example, rows are sorted on **manName**. Within groups containing matching **manName** values, rows are sorted on **stNum**.

The sort keys are used to define groups of rows. You can use the BEFORE GROUP and AFTER GROUP sections to take special actions on these groups.

One-Pass and Two-Pass Reports

Report data are processed in one of these two ways:

one-pass Rows are processed as they are produced.

Each time a row is produced by an OUTPUT TO REPORT statement, it is processed and the resulting output is written to the report destination.

two-pass Rows are collected, saved, sorted, and then processed.

As rows are produced, they are saved in a temporary table in the current database. When FINISH REPORT is executed, all the saved rows are retrieved in sorted order and processed.

4GL chooses between these methods based on two things: how the report rows are ordered, and how the report uses aggregate functions.

Two-Pass Logic for Row Order

Sorting of rows is essential to most reports. If the rows are not sorted, they cannot be divided into groups with similar values. These groups are the basis for subtotals and other summary information.

Sometimes it is convenient for the report driver to produce the rows in the sequence you need. Other times this is not possible; the rows are produced in random order. In such cases, **4GL** uses a two-pass report to sort rows before they are formatted.

Two-Pass Logic for Aggregate Values

Aggregate functions are used to get totals and other computed values (see "Using Aggregate Functions" on page 10-17). You use them to get totals and other computed values. You are allowed to refer to aggregate values anywhere in the FORMAT section of a report. When you use aggregate functions, you are asking for values based on the contents of all rows.

When you do not use aggregate functions, or when you use them only in the ON LAST ROW block, **4GL** can employ one-pass logic. But if you refer to aggregate functions outside the LAST ROW block, **4GL** must use a two-pass report. Here is how a two-pass report works:

- When the report driver executes OUTPUT TO REPORT, the row value is saved in a temporary table and the aggregate function values are accumulated in memory.
- When the report driver executes FINISH REPORT to indicate that no more rows will be produced, **4GL** retrieves all the rows from the temporary table in their proper sequence and sends them to the report for formatting.

The values of the aggregate functions are now available while the rows are processed, because they have been pre-computed.

Further Implications of Two-Pass Logic

When a report uses one-pass logic, the execution time of report output is distributed over all the OUTPUT TO REPORT statements because each row is formatted as it is received. The only action of FINISH REPORT is to print final totals.

When a report uses two-pass logic, the OUTPUT TO REPORT statement runs very quickly because it merely inserts a row in a temporary table. The actions of FINISH REPORT, however, can be quite lengthy because that is when all rows are retrieved and formatted.

A two-pass report builds a table in the current database. This table is created in the database that is current at the time START REPORT is executed. The same database must be open when OUTPUT and FINISH statements are executed. This places a restriction on the report driver: it cannot change databases (execute the DATABASE statement) during a two-pass report.

The FORMAT Section

Within the FORMAT section of a report, you place blocks of code that produce output lines of data in the report. The control blocks are named:

PAGE HEADER	Print the heading of any page
FIRST PAGE HEADER	Prints a special heading or cover page
PAGE TRAILER	Prints a footer at the end of any page
BEFORE GROUP	Initializes counters and totals at the start of a group of rows with similar contents; prints group headings

AFTER GROUP	Prints totals and other summary information following a group of rows with similar contents
ON EVERY ROW	Formats and displays detail lines. Accumulates totals and calculated values for use by AFTER GROUP blocks
ON LAST ROW	Displays final totals and aggregate values over all rows

4GL executes these control blocks automatically at appropriate times as rows are processed. For example, **4GL** calls the PAGE TRAILER code block when it is time to print the page trailer. When that block completes, **4GL** prints the blank lines corresponding to the BOTTOM MARGIN and prints the page-eject string, if any.

If more information is written to the report, **4GL** prints the TOP MARGIN blank lines and calls the PAGE HEADER block.

Contents of a Control Block

Within a formatting control block you can write any executable **4GL** statements you like. You can call functions; you can interact with the user; you can even start other reports and send output to them.

However, you must use caution when writing code that refers to global variables or interacts with the user. In a two-pass report, the formatting code is not called until all rows have been produced and FINISH REPORT has been executed. Global variables may not have the same values, and the screen may not display the same data, as when the rows were produced.

Formatting Reports

Usually a block contains code to test and set the values of local variables, and code to format and print the row values (the names defined in the REPORT statement argument list).

The following report execution statements are available to display data:

Statement	Usage
SKIP	Inserts blank lines
NEED	Causes a conditional page eject, so a set of lines can be made to appear on the same page
PRINT FILE	Embeds a file in the output
PRINT	Writes lines of data
PAUSE	Lets the user read the report (during output to screen only)

It is with PRINT that you send report data to the output destination. Like the DISPLAY statement, PRINT takes a list of values to display. Within a PRINT statement you can use the following keywords:

Keyword	Usage
COLUMN	Positions the data in the specified column
SPACES	Generates a calculated number of spaces
ASCII	Generates specific character values
USING	Formats numbers, including currency amounts
CLIPPED	Eliminates trailing spaces from CHAR values

PAGE HEADER and TRAILER Control Blocks

Within the PAGE HEADER and PAGE TRAILER control blocks, you write code that formats the pages of the report with fixed headings and pagination. The **minStockRep( )** report contains this page heading code (see page 10-6).

```
PAGE HEADER
    PRINT "Stock report",COLUMN 62,pNum USING "###"
    SKIP 2 LINES
    LET pNum = pNum + 1
```

It prints a fixed heading and a page number, and keeps count of the pages. A total of three lines is written, one line of heading and two blank lines. These lines are in addition to the TOP MARGIN lines specified in the OUTPUT section.

If a FIRST PAGE HEADER block is present, the PAGE HEADER block does not take control until the second page is started.

The **minStockRep( )** report contains this page trailer code:

```
PAGE TRAILER
    SKIP 2 LINES
    LET misc = 65 - LENGTH(thisMan)
    PRINT COLUMN misc,thisMan
    LET showManName = TRUE
```

The report prints the manufacturer name from the last-processed group of rows, right-justified, on the last line of the page. The person reading the report can find a manufacturer quickly by scanning the bottom right corner

of each page. The code also sets a flag that tells the ON EVERY ROW block to display the manufacturer name in the next detail line (because that will be the first detail line of the next page).

The FIRST PAGE HEADER section is similar to the PAGE HEADER except that **4GL** calls it only once, before any other block. You can use it to display a cover or a special heading on the first page. In a two-pass report you could put code in this section to notify the user that report output was finally beginning.

ON EVERY ROW Control Block

In the ON EVERY ROW control block you write the code to display each detail row of the report. Here is that block from the sample report specification on page 10-6.

```
ON EVERY ROW
    IF showManName THEN -- start of new group so...
        PRINT thisMan; -- with no newline
        LET showManName = FALSE
    END IF
    PRINT COLUMN 20, stNum USING "###" ,
        COLUMN 25, stDes CLIPPED ,
        COLUMN 45, stPrice USING "$,$$$.&&" ,
        COLUMN 55, stUnit CLIPPED
```

It displays a line like this:

```
      Nikolus            205  3 golf balls $312.00  case
```

The leading spaces are produced by the LEFT MARGIN statement in the OUTPUT section. This code suppresses the manufacturer name except in the first row of a group or the first row on a new page.

ON LAST ROW Control Block

After the final row has been processed and the FINISH REPORT statement encountered, **4GL** calls the ON LAST ROW control block. In this block, you can write code to summarize the entire report. You can use SKIP TO TOP OF PAGE in this block to ensure that the final totals appear on a new page. Here is the block from **minStockRep( )**:

```
ON LAST ROW
    SKIP TO TOP OF PAGE
    PRINT COUNT(*), " total rows processed."
```

Figure 10-3 A typical last row block

For the use of COUNT(*) and other aggregate functions, see "Using Aggregate Functions" on page 10-17.

BEFORE GROUP and AFTER GROUP Control Blocks

Whenever the value of a sort key changes between one row and the next, it marks the end of one group of rows and the start of another. In the **minStockRep( )** example, whenever there is a change of **manName**, a group of rows ends and another begins. The same is true of a change in **stNum**, but since stock numbers are unique in this particular situation, these "groups" never have more than one member.

A BEFORE GROUP control block is called when the first row of the new group has been received, but before it is processed by the ON EVERY ROW control block. In it, you can put statements that:

- Initialize counts, totals, and other values calculated group-by-group
- Print group headings, use the NEED keyword to ensure space on the page, or force a skip to a new page for the group
- Set flags and local variables used in the ON EVERY ROW block

Here is the BEFORE GROUP used in **minStockRep( )**, found on page 10-6:

```
BEFORE GROUP OF manName
    LET thisMan = manName
    LET showManName = TRUE
```

The first statement saves the manufacturer name from the first row of the new group so it can be used in the page trailer, as described earlier. The second statement sets a flag that tells the ON EVERY ROW block to display the manufacturer name in the next detail line (because that will be the first detail line of this group).

An AFTER GROUP block is called when the last row of its group has been processed by the ON EVERY ROW block. In it, you can put statements that calculate and print subtotals, summaries, and counts for the group.

Nested Groups

Each of the sort keys that you list in the ORDER section defines a group. In the example report there are two keys, and therefore two groups:

```
ORDER
    EXTERNAL BY manName,stNum
```

These groups are "nested" in the sense that the rows in a major group can contain multiple groups of the minor group.

In general, the BEFORE and AFTER blocks for minor groups will be called more times than those for major groups. The group for the last sort key you specify will change the most often. Only the ON EVERY ROW block will be executed more frequently.

Using Aggregate Functions

Often, reports that deal with sorted data need to collect aggregate values over the rows: counts, subtotals, averages, and extreme high or low values. You can produce such statistics yourself by writing code in different blocks. For example, you could collect an average value over a group this way:

1. In the BEFORE GROUP block, initialize variables for the sum and the count to zero.
2. In the ON EVERY ROW block, increment the count variable and add the current row's value to the sum variable.
3. In the AFTER GROUP block, calculate the average and display it.

4GL contains ready-made aggregate-value functions for most common needs. However, you can write code along these lines to collect unusual statistical values, to avoid the need to do two-pass reporting, or both.

Aggregate Calculations

For sums, averages (means), and extremes, **4GL** supplies these functions:

Function	Usage over many rows
SUM(*expression*)	accumulates a sum
AVG(*expression*)	calculates an average
MIN(*expression*)	finds a minimal value
MAX(*expression*)	finds a maximal value

The *expression* can be any **4GL** expression. Normally it will name one of the row values (one of the arguments to the REPORT), but it may also use constant values, local variables, and even function calls. (When using function calls or global variables, keep in mind that rows might all be processed at FINISH REPORT time.)

Any of these functions can be qualified with a WHERE clause to select only certain rows. The WHERE clause will usually test the row values themselves, but you can employ any test that is valid in an IF statement. For example, the following lines could be added to the LAST ROW block on page 10-6.

```
PRINT "Lowest item price", MIN(stPrice)
PRINT "Average low-cost item" ,
      AVG(stPrice) WHERE stPrice < 100
```

These lines would display the minimum over all **stPrice** values, and the average of all **stPrice** values that were less than $100.

Aggregate Counts

For counts, **4GL** supplies the COUNT(*) and PERCENT(*) functions. The value of COUNT(*) is the number of records processed by the report. You can see it in use in the ON LAST ROW control block on page 10-6. However, COUNT(*) can also be qualified with a WHERE clause so as to count only particular rows. The following could be added to the ON LAST ROW control block of **minStockRep( )**:

```
PRINT "number of boxed items" ,
     COUNT(*) WHERE stUnit = "box"
```

The PERCENT(*) function returns the value of one count as a percentage of the total number of rows processed:

```
PRINT "percent of case lots" ,
     PERCENT(*) WHERE stUnit = "case"
```

Aggregates Over a Group of Rows

You most often need aggregate values collected over the rows of one group, for example, a sum over a *group* to produce a *group* subtotal. You can use any of the six aggregate functions within an AFTER GROUP block for this purpose. You must use the word GROUP to specify that you want the aggregate value over the rows of the current group.

For example, the following lines could be added to the AFTER GROUP block in the **minStockRep( )** report on page 10-6:

```
PRINT "Count of stock items: " , GROUP COUNT(*) USING "<<<"
PRINT "Avg price of 'each' items: " ,
        GROUP AVG(stPrice) WHERE stUnit = "each"
```

This code displays a group count, which is simply a count of rows in that group, and an average. The average is taken over a subset of the rows of the group. When the subset is empty (when no rows have `stUnit="each"`) the value of the aggregate function is NULL.

The value of GROUP PERCENT(*) WHERE… may not be what you expect. It returns the number of rows in the group that met the condition, as a percentage of the total number of rows in the entire report (not as a percentage of the rows in the group). Because it requires the total number of rows, GROUP PERCENT forces a report to use two-pass logic. To calculate a percentage within a group, you can use explicit code such as:

```
PRINT "Pct 'each' items in group " ,
 ( (100 * GROUP COUNT(*) WHERE stUnit = "each")
    / (GROUP COUNT(*) )
USING "<<.&&"
```

Chapter

11

Using the Screen and Keyboard

Overview

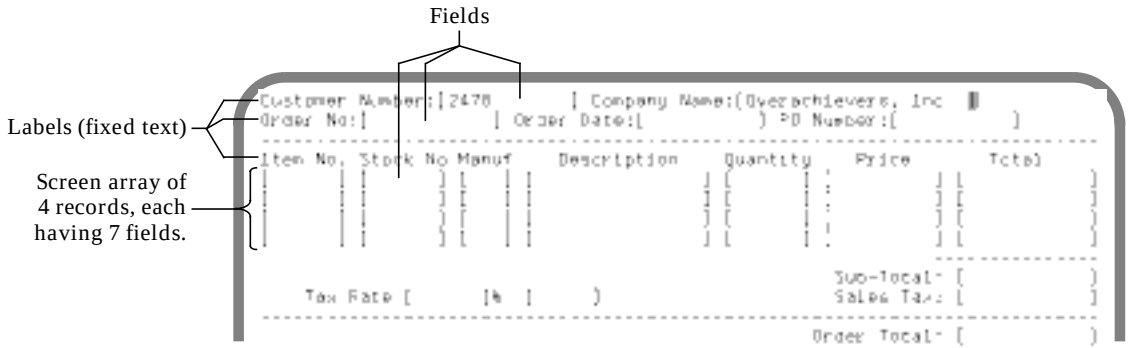
The architecture of the **INFORMIX-4GL** user interface is described in Chapter 3. The concepts behind the statements described in this chapter were covered in "Overview" on page 7-3.

This chapter details the way you program the user interface for your **4GL** applications. Since screen forms are very important to this, the first topic explains how you specify a form. Then statements you use to open **4GL** windows and fill them with forms and menus are discussed. At the end of the chapter, keyboard and screen line customization are discussed.

Specifying a Form

This section covers the contents of a form specification file in detail.The idea of managing screen forms in separate files was introduced in "Form Specifications and Form Files" on page 4-6. An overview of how your programs use forms is given in "How Forms Are Used" on page 7-11.

Here again is a screen image of a sample form:



4GL forms are traditionally designed in a WYSIWYG (what-you-see-is-what-you-get) environment using an ASCII text editor. Each ASCII form specification contains several sections. Some are mandatory, others are optional. The order of the sections is significant.

Form section	Usage
DATABASE	Specifies the database containing the tables and views whose columns are associated with fields in your form
SCREEN	Specifies the arrangement of fields and text that will appear in your form after it is compiled and displayed
TABLES	Specifies which tables have columns associated with the fields in the form, and can declare aliases for tables that require qualifiers
ATTRIBUTES	Declares a name for each field, and assigns attributes to it
INSTRUCTIONS	Specifies non-default delimiters and screen records

Chapter 5 of the *INFORMIX-4GL Reference* provides complete details on form specifications and statements.

The DATABASE Section

Typically a form field holds data from a particular column in the database. For example, the previous form begins with a field labeled **Customer Number**. This field displays data from the **customer_num** column in the **customer** table of the stores demonstration database.

You can associate a form field with a database column. In such cases, FORM4GL, the form compiler, will look in the specified database to determine the data type for that column and will use that as the data type of the form field.

The first line in the form specification file is a DATABASE specification. It tells the form compiler which database to use when looking for columns.

```
DATABASE stores2
```

Some forms use no database at all; that is, they have no fields that correspond to a database column. When this is the case, you can write:

```
DATABASE formonly
```

The SCREEN Section

In the SCREEN section, you specify the appearance of the form on the screen. With a text editor, you "paint" a picture of the form as you want it to appear. Here is the SCREEN section of the form on page 11-3. The lines between the curly braces (**{ }**) are a picture of the form as it is to appear on the screen. (The lines containing the braces are *not* part of the form image.)

```
SCREEN
{

Customer Number:[f000       ] Company Name:[f001                ]
Order No:[f002       ] Order Date:[f003      ] PO Number:[f004      ]
-------------------------------------------------------------------------
Item No. Stock No Manuf    Description    Quantity    Price       Total
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
[f005  ] [f006  ] [f07] [f008           ] [f009  ] [f010     ] [f011       ]
                                                              ---------------
                                                 Sub-Total: [f012       ]
    Tax Rate [f013  ]%  [f014 ]                  Sales Tax: [f015       ]
-------------------------------------------------------------------------
                                               Order Total: [f016       ]
}
```

The square brackets ([ ]) delimit *fields* into which the program can later display data and take input. Text outside brackets is *label* text; it will be displayed exactly as typed. A field consists of the spaces between brackets. If you need to make two fields adjacent, you can use a single vertical-bar symbol (|) to separate the fields.

Each form field must contain a *field tag*, a name used to link the field in the SCREEN section to its attributes in the ATTRIBUTES section of the form specification. The first field in the SCREEN section of the previous example has the field tag **f000**.

When you use the same field tag in more than one field (for example, the tag **f009** appears four times in the preceding example), you indicate that the field is really one part of a larger whole, either a *multiple-segment* field or a *screen array*.

Specifying Screen Dimensions

By default, the forms compiler assumes that your form will be displayed in a format of 24 lines of 80 columns each. If you intend to use the form in a larger window, you must specify larger dimensions in the SCREEN statement.

Screen Records and Screen Arrays

A *screen record* is a collection of fields that are treated as a group, just as a program record is a collection of members you want to treat as a related group. **4GL** makes it easy to display all the elements of a program record into the corresponding fields of a screen record, or to take input from a screen record into a program record.

A *screen array* is an array of similar fields that your program will treat as a unit, like a program array. **4GL** enables you to associate an array of data in the program with an array of fields on the screen, so that the user can view or edit the rows of data on the screen.

The example form above has a screen record composed of the fields tagged **f005**, **f006**, **f07**, **f008**, **f009**, **f010**, **f011**. These fields comprise an order item detail record. They are repeated four times vertically to make an array of four records. A statement in the INSTRUCTIONS section of the form (see "The INSTRUCTIONS Section" on page 11-11) is used to declare the array of records.

The process of displaying an array of data is covered under "Taking Input Through an Array" on page 11-27.

Multiple-Segment Fields

To display a string that is too long to fit on a single line of the form, you can create multiple-segment fields that occupy rectangular areas on successive lines of the form. The SCREEN section *must* repeat the same field tag in each segment of the multiple-segment field, and the ATTRIBUTES section must assign the WORDWRAP attribute to the field.

Your **4GL** program treats a multiple-segment field as a single field. When it displays text, any string longer than the first segment is broken at the last TAB or blank character before the end of the segment. Thus, word boundaries are respected. The rest of the string is continued in the first character position of the next segment. This is repeated until the end of the last segment, or until the last character is displayed, whichever happens first.

The TABLES Section

The TABLES section is closely related to the DATABASE section. In it you specify the tables that supply data for this form. The TABLES section of the form on page 11-3 is:

```
TABLES
    customer orders items stock state
```

Any tables containing columns named in the ATTRIBUTES section of the form (described in the next section of this chapter), must be listed in the TABLES section. The forms compiler will look for these tables in the database named in the DATABASE section.

You can also assign aliases to table names in the TABLES section. For example, in the TABLES section you can write:

```
TABLES postings = financial.postings
```

This gives the short table name **postings** to the table known in full as **financial.postings**. You *must* use the short name in the ATTRIBUTES section of the form.

Aliases are needed in the following cases:

- If you are utilizing an ANSI-compliant database. In such cases, you *must* declare table aliases—even if you are only drawing data from one table—*unless* the end-user of your application is also the owner of every table in the TABLES section. (This is because table owners must be specified when using ANSI-compliant databases.)
- If you are drawing data from several tables with the identical column names, since you cannot qualify a form field name with a table identifier or owner name.
- If you are accessing data from tables in remote databases, since you are restricted in the attributes section to the following format:

  ```
  table_identifier.column_name
  ```

- If you want a condensed way to refer to a table. The full name of a table can be quite lengthy since it can contain an owner name, a database name, and a site name. Aliases make referencing such tables more convenient and your form specification more readable.

Like the DATABASE section, the TABLES section is used only to get information needed by FORM4GL. It has no effect when the form is used by a program. The data you display into a form can come from any source, including any table in a Informix database.

The ATTRIBUTES Section

In the ATTRIBUTES section, you give detailed specifications for each of the fields you have defined in the SCREEN section. The two most important attributes for any field are its *field name* and its *data type*.

The field name is the name that you use, in your program, to put data into a field and get data out of it.

You assign these and other attributes in a series of specifications. Here is the attributes section for the form on page 11-3. It illustrates some of the many attributes that can be assigned to a field. (For a complete list of attributes, see Chapter 5 of the *INFORMIX-4GL Reference*.)

```
ATTRIBUTES
 f000 = orders.customer_num;
 f001 = customer.company;
 f002 = orders.order_num;
 f003 = orders.order_date, DEFAULT = TODAY;
 f004 = orders.po_num;
 f005 = items.item_num, NOENTRY;
 f006 = items.stock_num;
 f007 = items.manu_code, UPSHIFT;
 f008 = stock.description, NOENTRY;
 f009 = items.quantity;
 f010 = stock.unit_price, NOENTRY;
 f011 = items.total_price, NOENTRY;
 f012 = formonly.order_amount,TYPE MONEY(7,2);
 f013 = formonly.tax_rate;
 f014 = state.code, NOENTRY;
 f015 = formonly.sales_tax TYPE MONEY;
 f016 = formonly.order_total;
```

Figure 11-1 Attributes section from a form specification

The Field Name

A *field name* is the name your program uses to refer to a field for data display or input. The field name is often the same as the name of a database column. For example, here is the first specification from the sample attributes section:

```
f000 = orders.customer_num;
```

This says that the field shown in the SCREEN section with a field tag of **f000** is associated with the **customer_num** column of the **orders** table. As a result, the name of this field is **customer_num**. To display data into this field you would write a statement such as:

```
DISPLAY max_cust_num+1 TO customer_num
```

The Field Data Type

The data type affects how data is displayed in the field; for example, numeric data is right-justified and character data is left-justified. The data type also affects how the field behaves while the user is entering data during input; for example, the user cannot enter non-numeric characters in a field with a numeric type. You can specify the type of data that can be stored in a field directly or indirectly.

Fields Related to Database Columns

Very often you will want fields to display data taken directly from a database column. To simplify the definition of such fields, you can name the database column and table. The field then receives both its name and its data type from the database:

```
f009 = items.quantity ;
```

In this example, all of the fields with tag **f009** (there are four in the SCREEN section on page 11-5) take their name and data type from the **quantity** column of the **items** table. The **items** table must have been listed in the TABLES section of the form; also, a table of that name must exist in the database named in the DATABASE section at the time the form is compiled.

FORM4GL opens the named database and verifies that it contains an **items** table with a **quantity** column. It takes the data type of that column as the data type for the field. The advantage of this is that, if the database schema changes the data type of a particular column, you can keep your forms consistent with the new schema simply by recompiling them.

FORMONLY Fields

A field that is not related to a particular database column is called a *form only* field. You specify its name and its data type in the attributes statement:

```
f012 = formonly.order_amount TYPE MONEY(7,2)
```

The field with tag **f012** is given the name **order_amount**. In a program, you display data into the field using this name. Since FORM4GL cannot consult a database to see what the type should be, you must tell it with a TYPE attribute clause. If you omit information as to type, the default is a character string equal in size to the number of spaces between brackets in the SCREEN section.

Editing Rules

In the ATTRIBUTES section, you can specify editing rules for fields. Some of the rules you can apply to a field are shown in the following table (for complete details on form attributes see *INFORMIX-4GL Reference*, Chapter 5):

Attribute	Usage
DOWNSHIFT UPSHIFT	**4GL** changes the lettercase as the user types. An example of the UPSHIFT attribute can be seen in the ATTRIBUTES section shown on page 11-8.
INCLUDE	A list of specific values permitted on input. For example, you could limit entry to the letters "M" or "F," or to a certain range of numbers.
PICTURE	An edit pattern or *format string*, such as "(###) ###-####" for a USA telephone number. The user is allowed to type only letters or digits as the pattern dictates. The punctuation you supply such as parentheses or hyphens is filled in automatically as the user types.
REQUIRED	The user is not allowed to press Accept without having entered some value to the field.
VERIFY	**4GL** makes the user enter data to the field twice, checking that it is identical the second time.

You can also program specific input-editing rules by writing AFTER FIELD blocks in the INPUT statement.

Default Values

In the ATTRIBUTES section, you can specify a default value for a field. The value will be filled in automatically when an INPUT operation begins, and the user may replace it. (Alternatively, you can DISPLAY data into fields prior to input, and use the displayed data as default values.)

The INSTRUCTIONS Section

You use the INSTRUCTIONS section of the form for special features: field delimiters, screen records, and screen arrays.

Field Delimiters

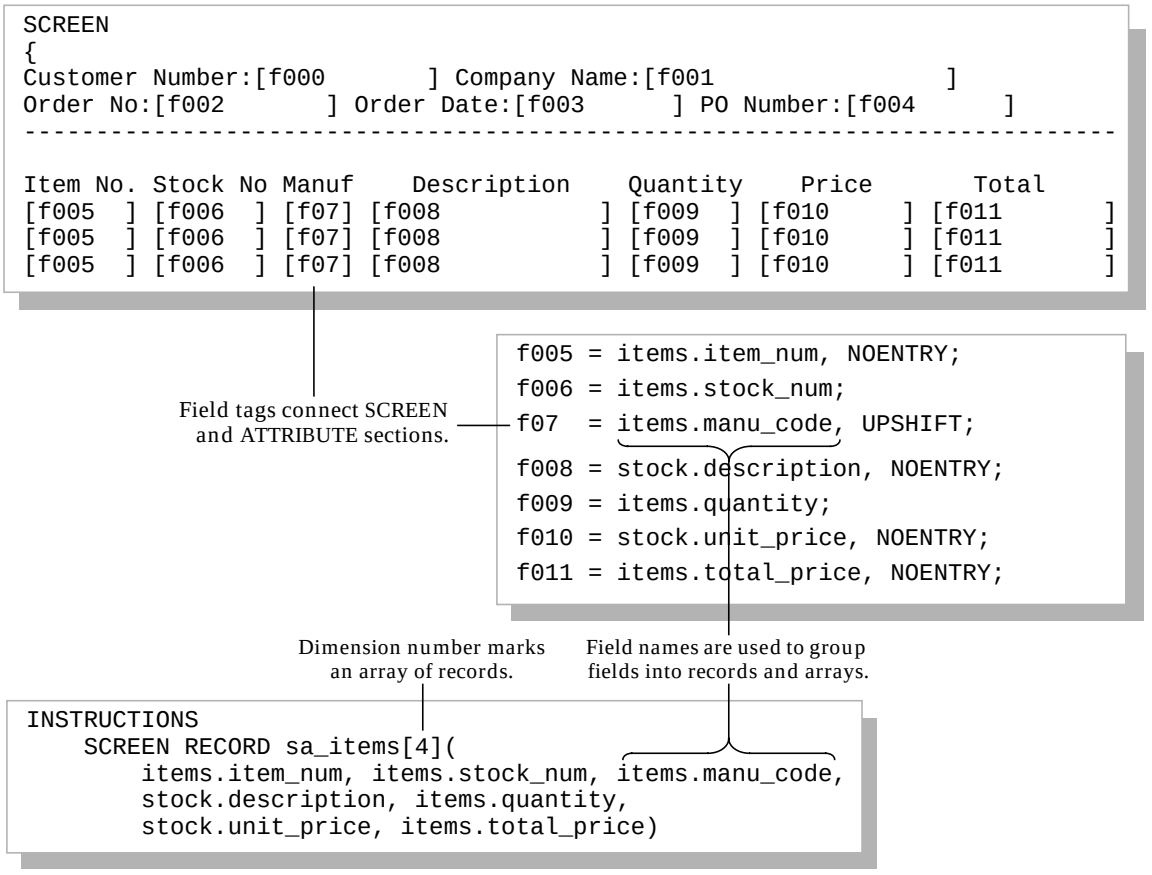
Normally when a form is displayed, fields are displayed with the same square brackets that you used in the SCREEN section to define them. In the INSTRUCTIONS section, you can specify different delimiters. Most often this feature is used to change the delimiters to spaces, thus causing the fields to be drawn without any visible markers.

Screen Records

A screen record is a group of fields that you want your program to treat as a unit. A screen record, like a program variable that is a record, is a collection of named members. You can display all of the members of a record variable into the matching fields of a screen record with a single DISPLAY statement. You can request input from all the fields of a screen record, and have the input values deposited in the matching members of a program variable, with one INPUT statement.

Screen Arrays

A screen array is a group of screen records that you want to treat as a scrolling table. The form shown on page 11-3 has a screen array defined this way. The following illustration shows the relationship between array fields as they appear in the SCREEN section, the ATTRIBUTES section, and the INSTRUCTIONS section of a form.



To use the array, your program must contain an array of records with similar members. It can use the DISPLAY ARRAY statement to display all the records in the array into the rows of the screen array.

Using Windows and Forms

For character-based terminals and workstations emulating them, all of your program's screen output takes place in a single window. If the output device is a real terminal, its screen is the window. If the output device is a graphical screen emulating a terminal in a graphical window, then the screen is *that* window.This section discusses the ways your program can make the best use of **4GL** windows.

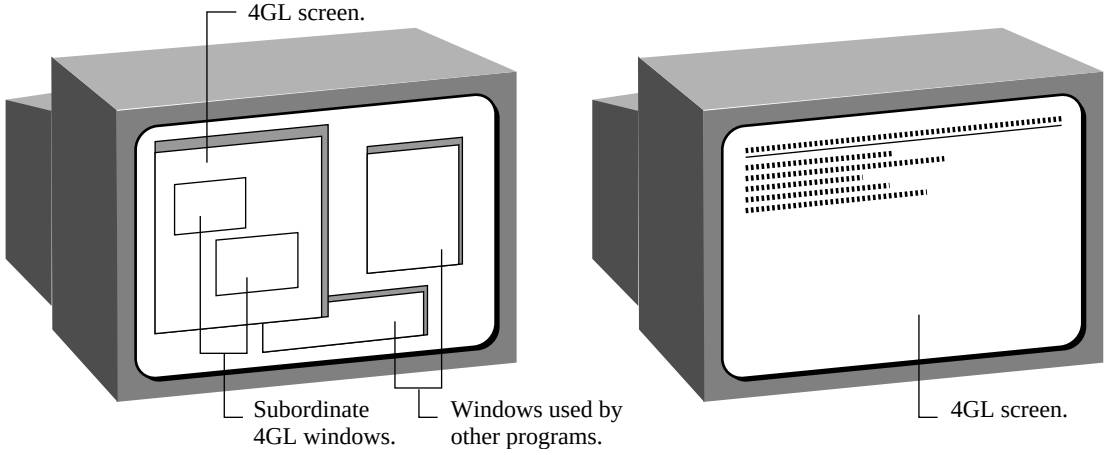


Figure 11-2 The 4GL screen on a workstation and a terminal

The program displays on the **4GL** screen. However, you can further control program output by creating additional rectangular areas within the confines of the screen. These can also be considered **4GL** *windows*. The size and contents of **4GL** windows within it, are under the control of your program.

An application can have many **4GL** windows. They can be the same size or smaller than the **4GL** screen. They can overlap. You can bring a particular window to the top, so that it is fully visible, or completely cover a particular window. And **4GL** windows can be closed and opened depending on the requirements of the application.

Opening and Displaying a 4GL Window

The initial **4GL** window is created automatically when **4GL** first encounters an I/O statement. This first **4GL** window fills the screen. In the statements you use for controlling windows, you can refer to this initial **4GL** window as SCREEN.

There is no way for the program to find out what that size is. Most programs assume that it allows 24 lines of 80 columns, since that is the size of most character-based terminals. If your program contains forms or reports that require a larger window, it may not run on some terminals.

Opening Additional 4GL Windows

You can open additional **4GL** windows with the OPEN WINDOW statement. This statement is covered in detail in Chapter 3 of the *INFORMIX-4GL Reference*. Here is an example of the statement:

```
OPEN WINDOW stockPopup AT 7, 4 WITH 12 ROWS, 73 COLUMNS
    ATTRIBUTE(BORDER, FORM LINE 4)
```

You can specify the following things when you open a new **4GL** window:

location	With the AT clause, you specify the location of the upper-left corner (1,1) of the new window in relation to the screen. The units are character positions.
size	In the WITH clause, you specify the size of the window in one of two ways: with a specific number of ROWS and COLUMNS, or by specifying a form. If you specify FORM or a WINDOW WITH FORM, the screen dimensions of that form establish the size of the new **4GL** window (see "Specifying Screen Dimensions" on page 11-5).
border	In the ATTRIBUTE clause, you can specify a border for the window. The statement above opens a bordered window.
color	In the ATTRIBUTE clause, you can also specify a color for all text displayed in the window. (You cannot specify a background color, only the foreground, or text, color.)
line numbers	You can set the locations of the standard lines for the menu, messages, prompts, and comments. These lines are discussed in detail under "Changing Screen Line Assignments" on page 11-29.

4GL Window Names

The first argument of OPEN WINDOW is a name for the window. You can use this argument to assign a global name to the window.

The following specifies **errorAlert** as a global name for a window:

```
OPEN WINDOW errorAlert AT 10, 20 WITH 4 ROWS, 40 COLUMNS
```

Anywhere else in the program (even in another source module), you can write a statement that refers to the **errorAlert** window. For example, here is how you would make a window current:

```
CURRENT WINDOW errorAlert
```

Controlling the Current 4GL Window

Only one **4GL** window can be current at a time. When you open a new **4GL** window, it becomes current. It is "on top" visually, covering any other **4GL** windows that it overlaps.

The current window receives all output of the DISPLAY, PROMPT, MENU, and MESSAGE statements. It is used by all INPUT statements. An error occurs if the program tries to use a form field when that field is not part of the form in the current window.

Clearing the 4GL Window

You can clear all displayed text from a window with the CLEAR WINDOW statement. This removes all output including menus, form fields, and labels. The window being cleared need not be the current window.

If you clear the current window from within a MENU statement, the menu will be redisplayed. This is not true of forms; you must redisplay a form explicitly.

Closing the 4GL Window

To remove a window, you use the CLOSE WINDOW statement. It makes the window invisible and unusable until you recreate it with OPEN WINDOW. When you close a window, the next window "below" it becomes the current window.

This means that if you are using a form or menu (and hence using the current window), and you call a subroutine that opens a window, uses it, and closes it again, the original window will again be current when the subroutine returns. This is the behavior you would expect.

However, if you are using a window and call a subroutine that makes another window current and does not close it, the wrong window will be current when the subroutine returns, and an error may follow.

Displaying a Menu

The key concepts of menus are covered in "How Menus Are Used" on page 7-8. That topic also includes an example of the code you use to create a menu. The details of the MENU statement are found in Chapter 3, *INFORMIX-4GL Reference*.

When you execute the MENU statement, a ring menu is displayed on the assigned MENU line of the current window. Normally, you will display a menu across the top line of a window, usually above the display of a form. But there are other ways to use menus.

```
FUNCTION alertMenu(msg , op1 , op2 , op3)
    DEFINE      windowWidth, indent SMALLINT ,
                ret , msg , op1 , op2 , op3 CHAR(20)
    LET windowWidth = 40
    LET indent = (windowWidth-LENGTH(msg))/2
    OPEN WINDOW alert2 AT 10,20 WITH 5 ROWS, windowWidth COLUMNS
        ATTRIBUTE (BORDER, MENU LINE LAST)
    DISPLAY msg CLIPPED AT 3,1
    MENU "Respond"
    COMMAND op1
        LET ret = op1   EXIT MENU
    COMMAND op2
        LET ret = op2   EXIT MENU
    COMMAND op3
        LET ret = op3   EXIT MENU
    END MENU
    CLOSE WINDOW alert2
    RETURN ret
END FUNCTION
```

Here the **alertmenu( )** function is given a short message and three choices. The choices would normally be keywords such as "OK," "No," "Cancel," or "Save." The function opens a small **4GL** window. In the window it displays the message line below a menu composed of the three choices.

The choices in the menu are not constants as is usually the case, but variables, specifically the function's arguments. Here is an example of how the function could be called:

```
MAIN
DEFINE ret CHAR(20)
CALL alertMenu("alert message here" , "FIRST" , "SECOND" , "THIRD")
    RETURNING ret
END MAIN
```

The function would display a window that looked like this:

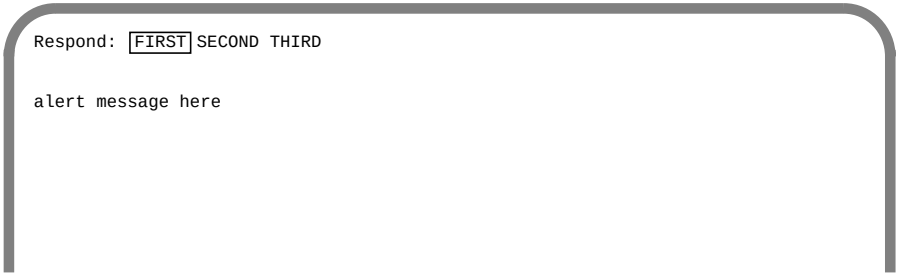


Figure 11-3 An alternative way of using the MENU statement

Whichever menu option the user chooses is returned as the function's result.

Note that although the function opens its window with the attribute MENU LINE LAST, the menu begins on the next-to-last line. See "Changing Screen Line Assignments" on page 11-29.

Opening and Displaying a Form

It takes only two program statements to open and display a form. The OPEN FORM statement brings the compiled form into memory using a command such as:

```
OPEN FORM orderFrm FROM "forders"
```

This statement causes **4GL** to search for a compiled form file named **forders.frm**. The file suffix **.frm** is supplied automatically and must not be used. **4GL** looks first in the current directory and then in directories named in the DBPATH environment variable.

Form Names and Form References

The first argument of OPEN FORM is a name for the form. You use that name to manage the form later in the program.

Specifying a Form Name

When you specify a form name in conjunction with the OPEN FORM statement, it becomes the global name of the form:

```
OPEN FORM orderForm FROM "forders"
```

Anywhere else in the program (even in another source module), you can write a statement that refers to this form.

Displaying the Form

After opening a form, you can use DISPLAY FORM to display it in the current window:

```
DISPLAY FORM orderForm
```

The current **4GL** window is cleared from the FORM line to the bottom. The fields and labels of the form are drawn on the current window starting at the FORM line. The fields are initially empty.

The form must fit within the current **4GL** window (see "Specifying Screen Dimensions" on page 11-5 and "Opening Additional 4GL Windows" on page 11-14), allowing for reserved lines.

By default, the Form line is the third line of the **4GL** window, but you can change it; see "Changing Screen Line Assignments" on page 11-29.

You can display a form repeatedly into one window or into different windows. However, you can only display one form per **4GL** window.

Displaying Data in a Form

When the form has been displayed, your program can display data in its fields. The DISPLAY TO statement is used for this. It takes a list of expressions and a list of field names in which to display them. You can display data to one field at a time.

```
DISPLAY "Salaried" TO emp_status
```

More often, you display a list of expressions in the form of variables to a list of fields.

```
DISPLAY theCustNum, theCustName, 0, 0
    TO  customer_num, company, order_amt, order_total
```

You can display all the elements of a record into the fields of a screen record with one statement.

A common technique is to use the names of database columns as the names of both the fields in a form and the members of a record. Here is how a record variable is defined to contain one member for each column in a table:

```
DEFINE custRow RECORD LIKE customer.*
```

If the form in the current window has a screen record with corresponding fields (see "Screen Records" on page 11-11) you can display all the members of this record variable into the fields of the screen record with one statement.

```
DISPLAY custRow.* TO custFields.*
```

Alternatively, when the current form has fields whose names are the same as the members of the record, you can display all the members this way:

```
DISPLAY BY NAME custRow.*
```

The BY NAME clause can be used whenever you want to display program variables into fields that have the same names.

Changing Display Attributes

4GL supports visual attributes such as REVERSE and BOLD, and a range of colors when the output device supports them. These display attributes can be assigned to a **4GL** window, to an entire form, or to one or more individual fields. Here are some techniques for making the best use of attributes:

- To set display attributes for the entire **4GL** application, use the OPTIONS statement before opening any windows.
- To set display attributes for all text in one window, use the ATTRIBUTE clause of the OPEN WINDOW statement.
- To set display attributes for all lines of a form without changing the attributes of other lines of the window, use the ATTRIBUTE clause of the DISPLAY FORM statement.
- To set specific display attributes for one field of a form, use the REVERSE or the COLOR clause for that field in the ATTRIBUTES section of the form specification file. The COLOR clause takes a WHERE keyword, so you can make the color of the field dependent on its contents or on the contents of other fields.
- To override the display attributes of specific fields as you display data into them, use the ATTRIBUTE clause of the DISPLAY TO statement.

These are the most important methods of setting visual attributes. There are others, and the precedence among methods is more complicated than this list shows. See Chapter 5, *INFORMIX-4GL Reference*, for complete information on visual attributes.

Combining a Menu and a Form

It is very common to have both a menu and a form in the same **4GL** window. The form fields provide the structure for displaying information. Your user can choose menu options to tell the program what information to display.

A common example is a program that lets the user browse through a series of rows from the database, one row at a time. Several of the example programs in *INFORMIX-4GL by Example* are devoted to just this problem. The basic technique is as follows:

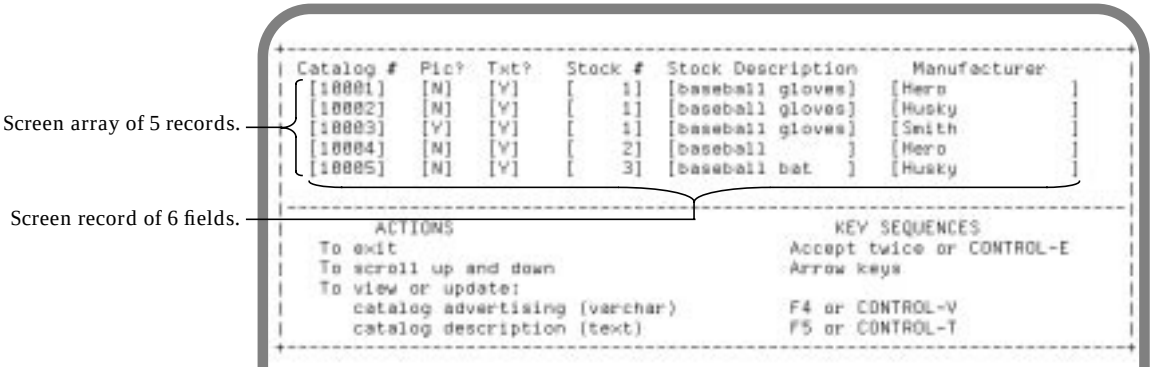
- Set up a database cursor to represent the selected set of rows
- Display a form that has fields for the columns of one row
- Execute a MENU statement that includes an option such as “Next” to cause the display of the next row

- In the COMMAND block for the "Next" menu option you use:
  - FETCH to get the next row from the cursor
  - DISPLAY to show the fetched values in the form

Thus each time the user selects menu option "Next," the program displays a new row. By using a SCROLL cursor (which supports backward as well as forward movement through the rows) you can easily provide menu choices for "Prior," "First," and "Last" row displays.

Displaying a Scrolling Array

Your **4GL** program may frequently need to display a scrolling list of rows. The screen array is used for this. In the following form, the rows of the **catalog** table from the **4GL** demonstration database are being scrolled through a screen array. (The form specification file for this form is from Example 18 of *INFORMIX-4GL by Example*.)



To implement a scrolling display like this your program must do two things:

- Display a form containing a screen array of screen records
- Define an array of records, each record having members that correspond to the fields of the screen records

In the example program, these two things are accomplished in the following manner.

Screen array of screen records is defined in the form specification file.

```
INSTRUCTIONS
    SCREEN RECORD sa_cat[5] (
        catalog_num,
        stock_num,
        manu_name,
        has_pic,
        has_desc,
        description)
```

```
DEFINE
        ga_catrows ARRAY[200] OF RECORD
            catalog_num    LIKE catalog.catalog_num,
            stock_num      LIKE stock.stock_num,
            manu_name      LIKE manufact.manu_name,
            has_pic        CHAR(1),
            has_desc       CHAR(1),
            description    CHAR(15)
        END RECORD
```

Array of records with matching names is defined in the program.

The screen array has five records; the array in the program has 200. The members of the records have the same names and appear in the same order.

The program uses a FOREACH loop to fill the array with rows fetched from the database (refer to "Row-by-Row SQL" on page 9-5). Once the program array has been loaded with data, the scrolling display can be started with the DISPLAY ARRAY statement. But first the program must call the built-in function SET_COUNT( ) to tell **4GL** how many records in the array have useful data. Only these rows will be shown in the display.

SET_COUNT( ) function tells **4GL** how many array items contain valid data.

ARR_CURR( ) function returns the index of the array row whose contents are in the current screen row.

```
CALL SET_COUNT(cat_cnt)
DISPLAY ARRAY ga_catrows TO sa_cat.*
    ON KEY (CONTROL-E)
        EXIT DISPLAY
    ON KEY (CONTROL-V,F5)
        CALL show_advert(arr_curr())
    ON KEY (CONTROL-T,F6)
        CALL show_descr(arr_curr())
END DISPLAY
```

During a DISPLAY ARRAY statement, **4GL** interprets the keystrokes used for scrolling (up and down arrows and others). It responds to them by scrolling the rows from the array in memory through the lines of the screen array.

As shown in the previous example, you can write ON KEY blocks within a DISPLAY ARRAY to act on specific keystrokes. In the example, if the user presses either CONTROL-V or F5, the program calls a function named **show_advert( )**. If you read Example 18 in *INFORMIX-4GL by Example*, you will see that this function opens a new window to display an expanded view of one column.

Taking Input Through a Form

Once your program displays a form, it can take input from the user through the form fields. In its simplest form, the INPUT statement, like the DISPLAY statement, takes a list of program variables and a list of field names.

```
INPUT stockNum, quantity FROM stock_num, item_qty
```

In this example, two fields—**stock_num** and **item_qty**—are enabled for input. Fields by these names must, of course, exist in the form displayed into the current window.

The program waits while the user types data into the field and presses the Accept key. (The Accept key is Escape by default, but the actual keyboard assignment can be changed; see "Run-Time Key Assignments" on page 11-31). The data is assigned into the program variables **stockNum** and **quantity** and the program proceeds.

INPUT supports the same shortcuts for naming records as DISPLAY does. You can ask for input to all members of a record, from all fields of a screen record and you can ask for input BY NAME from fields that have the same names as the program variables.

Help and Comments

In the ATTRIBUTES section of a form, you can specify an explanatory comment for any form field. These comments are displayed during input. When the cursor enters a field, the comment for that field is displayed on a specified line of the screen, the Comments line. (This line is by default the last line of the window, but can be changed; see "Changing Screen Line Assignments" on page 11-29.)

You can also associate a help message with any INPUT operation. See "How the Help System Works" on page 7-24.

Keystroke-Level Controls

Some programs require very precise control over user actions during input. You can do this, too, by writing one or more ON KEY blocks as part of the INPUT statement. **4GL** executes the statements in your ON KEY clause code block whenever the user presses one of the specified keys during input.

A typical use for an ON KEY block is to display special assistance. You can tell your user something like "Press CONTROL-P for a list of price codes." In an ON KEY block for the CONTROL-P key, you can open a **4GL** window and display in it the promised list. After getting the necessary information, your user can finish entering data and terminate the INPUT statement by pressing Accept or Cancel.

Field-Level Control

Sometimes you want to make a form even more responsive to the user or you may require more detailed control over the user's actions.

- To make a form seem lively and "intelligent," you want to cause a visible response to the user's last action, if possible anticipating the user's likely next action.
- To catch errors early, saving the user time, you want to verify each value as soon as it is entered, with respect to values in other fields.

You achieve this level of control by writing BEFORE FIELD and AFTER FIELD blocks of code as part of the INPUT statement. These are groups of **4GL** statements that are called automatically as the user moves the cursor through the fields on the form.

Using a BEFORE FIELD Block

A BEFORE FIELD block is executed as the cursor is just entering a field. In the following example a message is displayed when the cursor enters a field, and removed when the cursor leaves the field.

```
INPUT...
    ...
    BEFORE FIELD customer_num
        MESSAGE "Enter customer number or press F5 for a list."
    AFTER FIELD customer_num
        MESSAGE ""
```

You could get the same effect by writing the message as a COMMENT attribute for this field in the form specification. But if you did that, the message would be displayed whenever the form was used. In this case, the pop-up list of customers is a service offered only within this particular INPUT statement. The same form might be used in other contexts where you do not mean to do anything special for an F5 key.

A typical use of BEFORE FIELD is to prepare likely default values. Here, as the cursor enters a shipping-charge field, the program calculates, stores, and displays an estimated charge. This is done only if no value has previously been entered to the field.

```
BEFORE FIELD shipCharge
    IF shipRec.shipCharge IS NULL THEN
        LET shipRec.shipCharge =
                  shipEstCalc(shipRec.shipWeight,custRec.state)
        DISPLAY BY NAME shipRec.shipCharge
    END IF
```

Using an AFTER FIELD Block

An AFTER FIELD block is called as the cursor is just leaving a field. In it, you can write statements that:

- Check the value of the field for validity with respect to other form fields and with respect to the database
- Display values into other fields as a result of the value just entered into this one

Here is a simple example of validation:

```
AFTER FIELD customer_num
-- Prevent user from leaving an empty customer_num field
    IF gr_customer.customer_num IS NULL THEN
        ERROR "You must enter a customer number. Please do so."
        NEXT FIELD customer_num
    END IF
```

The NEXT FIELD statement sends the cursor to the specified field (in this case, back to the field it had just left). It terminates execution of the AFTER FIELD block and starts execution of the BEFORE FIELD block of the destination field.

The block from which the preceding example is taken (Example 15 in *INFORMIX-4GL by Example*) does more. When a customer number had been entered, it:

- Uses SELECT to read that customer's row from a database table.
- If no row exists, displays the fact and uses NEXT FIELD to repeat the input.
- Initializes other fields of the form with data from the database row.

Field Order Constrained and Unconstrained

In a BEFORE or AFTER FIELD block or an ON KEY block, you can also write a NEXT FIELD statement, forcing the cursor to move to a particular field. Many existing **4GL** programs control the cursor in this way. It is done to direct the user's attention to important data, or to require the user to enter certain data.

This tight control over actions of your user is a natural way to manage computer interaction on a character-based terminal. Programmers can assume that the cursor never enters the "discount" field without first having passed through the "customer number" field, for example. The BEFORE FIELD block for "discount" could therefore refer to the value of "customer number" with certainty that it was present.

Through the **4GL** OPTIONS statement you can adjust the amount of freedom your user will have in moving through a form. If you set FIELD ORDER CONSTRAINED, you can accurately predict the path your user will follow when moving through a form.

On the other hand, if FIELD ORDER UNCONSTRAINED is set, the user will be able to move through the form in any particular order using the arrow key.

Taking Input Through an Array

The DISPLAY ARRAY statement lets the user view the contents of an array of records, but the user cannot change them. You can use INPUT ARRAY to allow the user to alter the contents of records in the array, to delete records, and to insert new records.

The preparation for array input is very similar to that for DISPLAY ARRAY:

- You design a form containing a screen array of screen records.
- You define a program array of records. The members of each record match the fields of the screen record.
- If the user can alter existing data, you pre-load data into the program array and use the built-in **set_count( )** function to specify how many array items contain data.
- You execute an INPUT ARRAY statement, naming the program array as a receiving variable and the screen array as its corresponding field.

The same INPUT statement can also name ordinary program variables corresponding to ordinary form fields. The user directs the cursor through the fields as usual.

When the cursor enters the screen array on the form, **4GL** handles the scrolling of the array allowing the user to edit and change the contents of the fields, using arrow keys to navigate through the cells of the screen array. You can control and monitor these changes with BEFORE and AFTER FIELD blocks as usual. The built-in **arr_curr( )** function is available to tell you which array element is being changed. Besides the NEXT FIELD statement, you can execute the NEXT ROW statement to reposition the cursor to a different row.

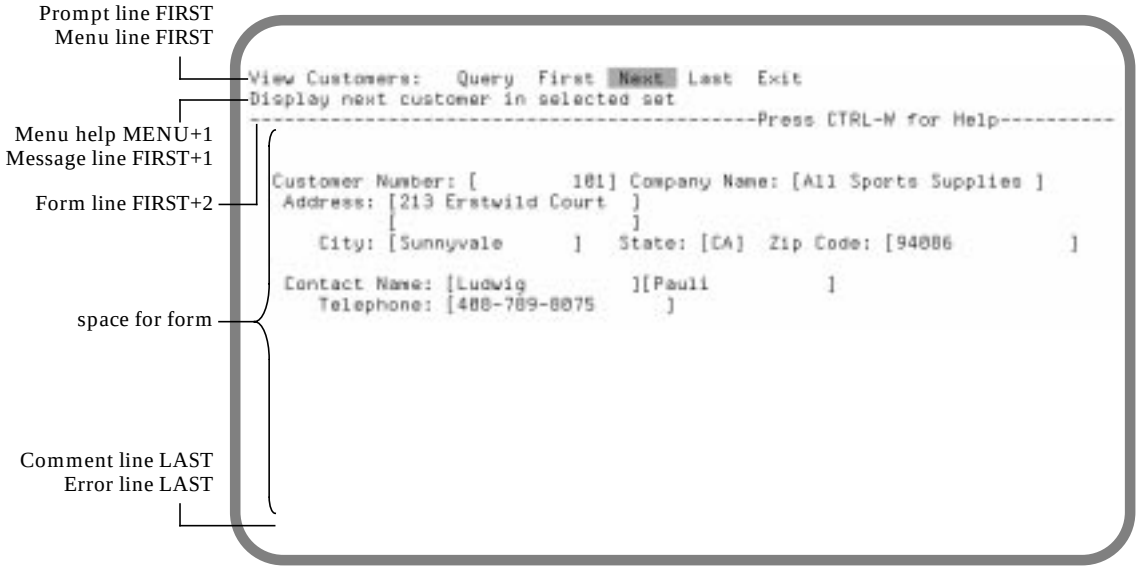
4GL also supports an Insert key to open a new, empty row in the array, and a Delete key to delete a row. You can monitor and control these actions with BEFORE and AFTER INSERT and DELETE blocks. You can redefine the Insert or Delete key using the OPTIONS statement.

Screen and Keyboard Options

Now that you understand how **4GL** uses the screen, here is a review of options for customizing the user interface.

Reserved Screen Lines

Here is a summary of the *reserved*, or *dedicated*, screen lines:



Prompt line	The PROMPT statement prompts the user for input to one or more variables. It displays its prompt string on the specified line and the user types a response on the same line.
Menu line	The MENU statement displays a "ring" (horizontal) menu on this line. The user moves the cursor from choice to choice with the Tab and Arrow keys. MENU and PROMPT can share the same line. Each clears the line and rewrites it as necessary.
Menu help line	In the MENU statement you can also write an explanatory phrase for each menu choice. As the user moves the cursor across menu options, the explanations are displayed on this line. The Menu line is always the line immediately below the menu.
Message line	The MESSAGE statement is used to show a short message to the user. The Message line can share the line after the Menu because the MENU statement rewrites its help line when it starts or resumes. The Message line could also be the same as the Comments line.

Form line	The *top line* of a screen form will be displayed on this line by the DISPLAY FORM statement.
Comments line	When designing a screen form, you can specify an explanatory comment for any field. As the user moves the cursor through the fields of the form, these explanations are displayed on the Comments line.
Error line	The ERROR statement is used for providing the user with a message serious enough to warrant an audible warning. The Error line is special in another way. All other dedicated lines represent positions within the current **4GL** window where text can be written, replacing previous text on that line. The Error line specifies a position where a one-line error message appears on the screen. In other words, the Error line appears on the specified line of the screen without regard to the position or size of the current **4GL** window.

Changing Screen Line Assignments

The default layout of reserved screen lines is usually satisfactory. If you change it, you should make sure that your new layout is consistent across your application and consistent with other applications that your users will see.

You can change the assignment of logical lines to line numbers in two ways, depending on your needs:

- With the OPTIONS statement, you can change the assignment of one or more lines for all windows.
- With the ATTRIBUTES clause of OPEN WINDOW, you can assign the logical lines for one window when you create it.

There are, however, some line assignments you cannot change. For example, the Menu Help line always follows the Menu line, so the Menu line can never be LAST. If you specify MENU LINE LAST, **4GL** treats it as if you had assigned LAST-1.

Getting the Most on the Screen

The most common reason for changing screen line assignments is to increase the maximum number of lines available for other purposes. To do this, you make multiple screen lines use the same row. The following table shows which lines can share the same screen row:

	Menu	Menu help	Message	Form	Comment	Error
Prompt	yes	yes	yes	note 1	yes	yes
Menu		no	note 2	note 1	note 3	yes
Menu help			note 2	note 1	yes	yes
Message				note 1	yes	yes
Form					note 1	yes
Comment						yes

Figure 11-4 Table shows which menu lines can share the same screen row

In this table, the word "yes" means that the two intersecting types of screen lines can share the same screen row because the lines appear at different times, and each clears the row before using it. Potential problems are discussed in the following notes:

Note 1 The screen form is not automatically redrawn after it has been overwritten. If you display a prompt, menu, message, or comment into a line used by a form, the only way to restore the complete form is to redisplay the form and then redisplay the data in its fields.

Note 2 If you make the Message line the same as Menu or Menu help, you must be careful when using the MESSAGE statement from within a MENU. You must program a delay before resuming the menu operation. Otherwise the menu will replace the message text too quickly for the user to read it. (If messages and menus are used at different times, there is no difficulty about them using the same row.)

Note 3 When you use both a menu and a form you should probably not make the Comments and Menu lines the same. You can make Comments the same as Menu help. Then both types of explanations appear on the same screen row.

The ERROR text is always displayed on the designated Error line of the physical screen. When you design a **4GL** window as described under "How 4GL Windows Are Used" on page 7-21, you do not need to allow for an Error line.

Run-Time Key Assignments

The **4GL** run-time environment uses several logical function keys and provides default keyboard assignments. These can easily be reassigned. The abstract function keys are summarized in the following table:

Key Name	Purpose of Key	Default Keystroke
Accept	Selects the current menu option in a statement; terminates input during CONSTRUCT, INPUT, and INPUT ARRAY; terminates DISPLAY ARRAY.	Escape
Interrupt	Represents the external interrupt signal; available when interrupts are deferred with the DEFER statement.	stty interrupt key (usually CONTROL-C)
Insert	Requests insertion of a new line during INPUT ARRAY, starting execution of a BEFORE INSERT block.	F1
Delete	Requests deletion of the current line during ARRAY, starting execution of a BEFORE DELETE block.	F2
Next	Causes scrolling to the next page (group of lines) during DISPLAY ARRAY and INPUT ARRAY.	F3
Previous	Causes scrolling to the previous page (group of lines) during DISPLAY ARRAY and INPUT ARRAY.	F4
Help	Starts the display of the specified HELP message from the current help file.	CONTROL-W

Figure 11-5 Logical command keys in the 4GL run-time environment and their default assignments

You can change the assignment of the logical keys to physical keystrokes with the OPTIONS statement. There are two common problems that require you to change them.

- The Escape key is often a prefix value for function keys. The operating system may wait a fraction of a second after the Escape key is pressed, in order to make sure it is not the start of an escape sequence, before passing it to the program. On some systems this can cause a delay in the response of your program to the Accept key.
- The numbering of function keys is not consistent from one version of UNIX to another. This is because some terminals may have different physical keys than those defined in **termcap** file.

Dedicated Keystrokes

The following physical keys have dedicated uses during some **4GL** statements:

Key Name	Use in INPUT, INPUT ARRAY, and CONSTRUCT	Use in MENU
CONTROL-A*	Switches between overtype and insert modes.	None.
CONTROL-D*	Deletes from the cursor to the end of the field.	None.
CONTROL-H* (backspace)	During text entry, moves the cursor left one position (nondestructive backspace).	Moves highlight to next option left.
CONTROL-I* or TAB*	Cursor moves to next field; except in a WORDWRAP field, inserts a tab or skips to a tab depending on mode.	None.
CONTROL-J (Linefeed)	Cursor moves to next field; except in a WORDWRAP field, inserts a newline or moves down one line, depending on mode.	Moves the highlight to the next option right.
CONTROL-L*	During text entry, moves the cursor right one position.	Moves the highlight to the next option right.
CONTROL-M or Return	Completes entry of the current field. Cursor moves to next field if any; else same as Accept.	Accepts the option that is currently highlighted
CONTROL-N	Same as CONTROL-J	None.
CONTROL-R*	Causes the screen to be redrawn.	Causes the screen to be redrawn.
CONTROL-X*	Deletes the character under the cursor.	None.
Left Arrow	Same as Backspace.	Same as Backspace.
Right Arrow	Same as CONTROL-L.	Same as CONTROL-L.
Up Arrow	Usually moves to previous field; except in a WORDWRAP field moves up one line in field and in an INPUT ARRAY moves to the corresponding field in the previous row.	Moves the highlight to the next option left.
Down Arrow	Usually moves to next field; except in a WORDWRAP field moves down one line in field and in an INPUT ARRAY moves to the corresponding field in the next row.	Moves the highlight to the next option right.
Insert	Same as logical INSERT key	None.

Figure 11-6 ***Effect of special keys on interactive statements and within menus.***
An asterisk indicates that the key cannot be used in an ON KEY clause.

Key Name	Use in INPUT, INPUT ARRAY, and CONSTRUCT	Use in MENU
Delete	Same as logical DELETE key	None.
PgUp	Same as logical NEXT	None.
PgDn	Same as logical PREVIOUS	None.

Figure 11-6 ***Effect of special keys on interactive statements and within menus.***
An asterisk indicates that the key cannot be used in an ON KEY clause.

Intercepting Keys with ON KEY

The names shown in the first column of the preceding two tables are accepted in an ON KEY clause with the noted exceptions. If you intercept these keys using ON KEY, they lose their dedicated abilities. For example, if you intercept the UP key in a DISPLAY ARRAY statement, the user will have no way to move the cursor upward. If you name RIGHT in a KEY clause of a MENU COMMAND, the user will not be able to move through the menu using the right-arrow key.

Chapter

12

Handling Exceptions

Overview

Exceptions, usually referred to as *errors*, are unusual occurrences that you might sometimes wish would never happen to your program. Of course, you know they *will* happen, and you know you need to write your programs so they behave in reasonable ways when errors and other unplanned for events occur.

This chapter reviews the categories of exceptional conditions and how **4GL** reacts to them when you do not specify what to do. Then it details the mechanisms that **4GL** gives you for handling them:

DEFER — Allows you to convert asynchronous signals into synchronous flags that you can poll.

WHENEVER — Lets you change how **4GL** responds to specific error conditions.

Error type	**Check**
Interrupt signal	int_flag
Quit signal	quit_flag
Run-time error, expression error, file error, display error, initialization error	status
SQL error	status, SQLCA.SQLCODE
SQL warnings	SQLCA.SQLAWARN
SQL end of data error	status=NOTFOUND *or* SQLCA.SQLCODE=NOTFOUND

Figure 12-1 Common error conditions and the built-in flags they set

Exceptions

You can write your **4GL** program to recognize and respond to the following types of exceptions:

- Run-time errors (sometimes called *execution errors*)
- SQL *end of data* conditions
- SQL warnings
- Asynchronous signals, meaning signals from the keyboard or elsewhere, occurring at an unplanned for time

Run-Time Errors

Run-time or execution errors are serious conditions that are detected by the database engine or **4GL** during run-time. Although they are divided into several categories, these errors all have one thing in common: if they occur and the program does not explicitly handle them, the program will terminate immediately. To handle these types of errors, you must use the WHENEVER ERROR statement. See "Using the WHENEVER Mechanism" on page 12-10 for more information.

A negative error code number is associated with every type of execution error. For every number there is an error message. Error numbers and their associated messages are available on-line. (See the introduction of *INFORMIX-4GL Reference*.) In addition, your **4GL** application can call a function, **err_get( )**, to retrieve the message text for any error number. See the chapter on built-in functions and operators in Chapter 4 of the *INFORMIX-4GL Reference*.

Execution errors are divided into five groups based on the kinds of program actions that can cause them:

Expression errors arise when **4GL** attempts to evaluate an expression that violates the rules of the language. For example, error −1348 occurs when an attempt is made to divide by zero. Error −1350 occurs when **4GL** cannot convert between data types.

File errors arise when **4GL** tries to access a file and the operating system returns an error code. For example, error −1324 means that a report output file could not be written.

SQL errors arise when the database engine detects an error in an SQL statement. For example, error −201 results from a syntax

	error in an SQL statement, while error –346 shows that an attempt to update a row in a table failed.
screen errors	arise when something goes wrong with a screen interaction. For example, error -1135 means that the row or column in a DISPLAY AT statement falls outside the current **4GL** window.
initialization and validation errors	The INITIALIZE and VALIDATE statements are used to initialize or test program variables against a special database table, **syscolval**. Errors in the operation of these statements are in a separate category; you handle them apart from other errors.

SQL End of Data

When the database engine is unable to retrieve a specified row, it reaches an end of data condition and sets the SQLCODE member of the SQLCA record to 100. **4GL** includes a built-in constant called NOTFOUND that has a value of 100.

By default, **4GL** continues execution of your program when an End of Data condition is encountered. However, you can test for this condition after the following statements:

- FETCH
- FOREACH
- SELECT

If the value of **sqlca.sqlcode** is greater than zero, there was no row available.

Alternatively, your program can treat end of data as an error condition so that when it occurs, your program is diverted to code that handles end of data.

To change the default behavior of **4GL**, use the WHENEVER NOT FOUND statement. (See "Using the WHENEVER Mechanism" on page 12-10 for more information.)

SQL Warnings

Some SQL statements can detect conditions that are not errors, but may provide important information for your program. These conditions are called *warnings* and are signalled by setting warning flags in the SQLAWARN member of the SQLCA record.

As with end of data, you have a choice in how to treat an SQL warning. By default, **4GL** sets SQLCA.SQLAWARN and continues execution of the program. You can insert code after any SQL statement to test the SQLAWARN flag values.

To change this default behavior, you can use the WHENEVER WARNING statement, described beginning on page 12-10.

Asynchronous Signals: Interrupt and Quit

External signals are *asynchronous* signals that—unless specifically *deferred*—are delivered by the operating system to a running program. An asynchronous signal is one that is not related to an action of the program.

Common external signals such as Interrupt and Quit would typically be generated by the user. The keystroke that generates an Interrupt is known as the Interrupt key; the keystroke that generates a Quit is known as the Quit key. (On some systems, physical keys associated with external signals can be reassigned.)

Generally speaking, unless intercepted, an external signal that reaches an application causes the application to terminate immediately. **4GL** provides mechanisms for testing for and handling Interrupt and Quit signals.

Uses of Asynchronous Signals

A first glance, it is difficult to see why an external signal of any kind should be allowed to terminate a running program. But in fact, it is often quite useful to allow user-induced exceptions to end programs during certain stages of program development or debugging.

For example, if you create a FOREACH or WHILE routine without a termination point, you cannot stop the routine without killing the process or rebooting the system. However, if interrupts are not trapped, pressing the Interrupt key (CONTROL-C by default) ends the program immediately. Later, once you have perfected the routine, any external signals can be deferred using the **4GL** DEFER statement (described in the next section) and more polite mechanisms put in place to handle unanticipated user requests.

Using the DEFER Statement

Since it is generally not convenient for the user to immediately terminate an application from the keyboard, **4GL** provides a DEFER statement that:

- Captures an Interrupt or Quit signal.
- Sets the appropriate **4GL** flag, allowing you to deal with the external signal programmatically.

The DEFER mechanism allows you to choose how to handle Interrupt or Quit signals. To enable this mechanism, you include the DEFER statement once, at the beginning of your **4GL** program. This statement has two forms:

DEFER INTERRUPT	the Interrupt key combination is trapped and TRUE is assigned to the built-in global integer variable **int_flag**. Once deferred, the Interrupt signal causes no effect on most program statements; however it does terminate some interactive statements, as described in the next section.
DEFER QUIT	the **4GL** Quit key combination, CONTROL-\, is caught and TRUE assigned to the built-in global variable **quit_flag**.

Note: *Some systems may also be able to deliver other external signals, but these are not handled by the* **4GL** *DEFER mechanism. There are also synchronous signals that represent program run-time errors trapped by the operating system, usually for major program faults such as indexing past the end of an array.*

The DEFER statement can only be executed in the MAIN program block. Its effect cannot later be undone. Once deferred, Interrupt can be named as a logical key in an ON KEY clause.

Interrupt with Interactive Statements

The effect of a user-generated Interrupt when DEFER INTERRUPT is in effect can be summarized as follows:

INPUT	When INTERRUPT is named in an ON KEY clause, the signal is treated as just another keystroke. In other words, you can trap an Interrupt signal. If the signal is not trapped, it causes the INPUT to end. The AFTER INPUT block is executed, if one exists; then, control goes to the next statement.
CONSTRUCT	When INTERRUPT is named in an ON KEY clause, the signal is treated as the activation key for that control block. Otherwise, the signal causes CONSTRUCT to end. The AFTER CONSTRUCT block is executed, if one exists; then, control goes to the next statement.

DISPLAY ARRAY	When INTERRUPT is named in an ON KEY clause, the signal is treated as just another keystroke. Otherwise, the operation ends and control goes to the next statement.
MENU	When INTERRUPT is named in a MENU clause, the signal is treated as just another keystroke. The menu operation continues. However, INTERRUPT can be named in a COMMAND KEY list and used to initiate an action.
PROMPT	When INTERRUPT is named in an ON KEY clause, the signal is treated as just another keystroke. In other words, you can trap an Interrupt signal. When an Interrupt is generated, the operation ends and NULL is assigned to the receiving variable.

Interrupt with INPUT and CONSTRUCT

You will often use program logic similar to the following program fragment with INPUT or CONSTRUCT statements:

```
    DEFER INTERRUPT
    LET int_flag = FALSE
    LET cancelled = FALSE
-- the Interrupt flag is specifically set FALSE
    INPUT ...
        BEFORE INPUT
            MESSAGE "Use CTRL-E to cancel input."
        ON KEY (CONTROL-E)  -- logical cancel
            LET cancelled = TRUE
            EXIT INPUT
        ...other clauses of INPUT...
        AFTER INPUT
-- tests to see if an Interrupt has been received (is TRUE)
            IF int_flag THEN
                LET cancelled = TRUE
            ELSE
                ...post-process based on Accept key...
            END IF
    END INPUT
    IF cancelled THEN
        MESSAGE "Input cancelled at your request!"
    END IF
```

The code sample establishes three ways for the user to terminate the INPUT operation.

Accept key	Normal completion of INPUT is signaled by pressing the Accept key. The statements in the AFTER INPUT block are used to perform any final validation of the entered data.
Interrupt key	When the user generates an Interrupt, **4GL** ends the INPUT operation, but in doing so it executes the AFTER INPUT block. The program checks the setting of the **int_flag**. If TRUE, the INPUT operation is terminated. It exits the AFTER INPUT block early if it was entered due to an Interrupt.
CONTROL-E	The program establishes a logical Cancel based on using an unassigned control key trapped by an ON KEY block.

The example code converts both cancellations, the built-in cancellation due to Interrupt and the programmed signal based on CONTROL-E, into a TRUE value in a variable named **cancelled**. This is cleared before the INPUT operation begins, and is tested afterward.

When Interrupt is used in an ON KEY clause, Interrupt no longer terminates the operation, so the preceding example could be written this way:

```
-- DEFER INTERRUPT run earlier
    LET cancelled = FALSE
    INPUT ...
        BEFORE INPUT
            MESSAGE "Use ctrl-E to cancel entry"
        ON KEY (control-E, Interrupt) —— logical cancel
            LET cancelled = TRUE
            EXIT INPUT
        ...other clauses of INPUT...
    END INPUT
    IF cancelled THEN
        MESSAGE "Input cancelled at your request!"
    END IF
```

This explicit handling of Interrupt with an ON KEY statement prevents execution of AFTER INPUT following an Interrupt. Logic of either type can also be used with CONSTRUCT.

Deferred Interrupt with the MENU Statement

Normally the arrival of the Interrupt signal has no effect on a MENU operation. Typically a program should treat a user-generated Interrupt key as a sign that the user is at least impatient. You can write a MENU program block so that it treats Interrupt as a keystroke and uses it to exit the menu:

```
MENU "Scrolling menu"
COMMAND "First" "View first row of set" HELP 301
    ...
COMMAND "Next" "View next row of set" HELP 302
...etc etc
COMMAND KEY(ESCAPE, INTERRUPT) "Exit" "Exit this menu"
    EXIT MENU
END MENU
```

Using the WHENEVER Mechanism

The WHENEVER statement handles:

- Run-time errors
- SQL warnings
- SQL end of data conditions

The WHENEVER statement is based on the ANSI standard for embedded SQL, which defines the keywords and basic operation of that statement.

What WHENEVER Does

The form of the statement is:

WHENEVER *event action*

You use this mechanism to tell **4GL** that when a certain *event* occurs, a certain *action* should be done. Following WHENEVER you write the normal line of program logic, just as if the errors would never happen. (For full details on using WHENEVER, see Chapters 3 and 4 of the *INFORMIX-4GL Reference*.)

Actions of WHENEVER

You can specify four possible types of actions using variations of the WHENEVER statement:

do not terminate — You write WHENEVER…CONTINUE to specify that when the error happens, it is to be ignored. **4GL** sets the **status** variable and continues executing the program.

terminate — You write WHENEVER…STOP to specify that when the error happens, the program is to be terminated. Any uncommitted database transaction is rolled back and an error message is displayed.

call a function — You write WHENEVER…CALL *function* to say that, when the error occurs, the named *function* should be called.

go to a label — You write WHENEVER…GOTO *label* to say that, when the error occurs, the program is to branch to a certain *label*.

Note: *The appearance of WHENEVER in a source module changes the way the compiler produces code, starting with that line and continuing through the source code to the end or to the next WHENEVER. When you specify an action other than do nothing (CONTINUE), the compiler automatically generates code to test for an error following each statement that might cause one, and to carry out the specified action when an error occurs.*

Errors Handled by WHENEVER

You use the first argument of WHENEVER to specify the error or errors to be handled. The keywords and the errors trapped are shown in this table.

Keyword	End of Data	SQL Warning	SQL Errors	Screen Errors	Initialize and Validate	Expression Errors	File Errors
NOT FOUND	■						
WARNING or SQLWARNING		■					
SQLERROR			■				
ERROR			■	■	■		
ANY ERROR			■	■	■	■	■

Note: *The* **4GL** *C code compiler traps expression errors with either ERROR or ANY ERROR set.*

By default, **4GL** executes a program as if it contained the following WHENEVER statements:

```
    WHENEVER NOT FOUND CONTINUE
    WHENEVER SQLWARNING CONTINUE
    WHENEVER ANY ERROR STOP
```

That is, warnings and end of data exceptions are ignored while all errors cause the program to end.

Using WHENEVER in a Program

The following function contains two SQL statements, either of which could produce an error if the arguments of the function are not correct. (The purpose of the function is discussed in "Dynamic SQL" on page 9-10.)

To prevent run-time errors in these two statements from terminating the program, the WHENEVER statement has been inserted, along with code to test the SQL error value after each statement.

```
FUNCTION tableGrant( )
    DEFINE whom , tab , priv , granText VARCHAR(255)

    LET granText = "GRANT " , priv,
        " ON " , tab ,
        " TO " , whom
WHENEVER SQLERROR CONTINUE
    PREPARE granite FROM granText
    IF SQLCA.SQLCODE == 0 THEN
        EXECUTE granite
        IF SQLCA.SQLCODE == 0 THEN
            RETURN -- all done in good style
        END IF
    END IF
WHENEVER SQLERROR STOP
    -- Some SQL error if we get here
    DISPLAY "The following prepared statement is in error:"
    DISPLAY granText
    DISPLAY "Error is ",SQLCA.SQLCODE USING "-<<<<<<",
        ", meaning:"
    DISPLAY ERR_GET(SQLCA.SQLCODE)
END FUNCTION
```

The WHENEVER statements are placed closely around the statements of interest. The WHENEVER SQLERROR CONTINUE statement causes errors in SQL statements to be ignored. The only effect of an error will be to set a negative code in the SQL communications area and the **status** variable.

The WHENEVER SQLERROR STOP statement restores the default handling of SQL errors. If it was omitted, the effect of the first statement would continue to the end of the source module.

Using WHENEVER ERROR for Non-Fatal Errors

You can prevent most execution errors from terminating the program using the WHENEVER ERROR statement, which establishes your program's policy for handling execution errors. For example, when an error is encountered at a particular point in your program you could:

- Ignore errors
- Call a function
- Jump to a label
- Display a PROMPT statement to get more guidance from your user

The WHENEVER ERROR statement is a condensed way of putting an IF statement after every SQL statement in order to establish what action should be taken when errors occur.

Using WHENEVER ANY ERROR for Expression Error

You can cause a variable to be checked after an expression error by using the ANY keyword with a WHENEVER ERROR statement.

Using WHENEVER WARNING for SQL Warnings

By default, **4GL** sets SQLCA.SQLAWARN and continues execution when it encounters a warning. Although the statement WHENEVER WARNING can be used to make the program call a function or go to a label when an SQL warning flag is set, normally you do not want the SQL warning flags to divert the program.

Using WHENEVER NOT FOUND for SQL End of Data

By default, **4GL** sets SQLCA.SQLCODE (and status to 100 (NOTFOUND)) and continues execution when it encounters an end of data condition. Although the statement WHENEVER NOT FOUND can be used to make the program call a function or go to a label when SQL input finds end of data, you normally do not want an "end of data" to cause any diversion of the program.

Notifying the User

The **4GL** Error line is a one-line display dedicated to the display of messages. (See "Screen and Keyboard Options" on page 11-27.) The Error line becomes visible in three cases:

- When you execute the ERROR statement to display a message that you composed.
- When you execute the ERR_PRINT( ) function to display a message based on a **4GL** error number.
- When you execute the ERR_QUIT( ) function to display a message and terminate the program.

In fact, the Error line is a one-line **4GL** window that is automatically opened on the screen (see "Opening and Displaying a 4GL Window" on page 11-14). After an ERROR statement or call to ERR_PRINT( ), this window remains open and visible until a keystroke event occurs. Then it is closed, revealing any form or menu that might be hidden beneath it.

Index

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W