

Querix 4GL Reference



Querix 4GL Reference
Version 2.2 – September 2014
Revision number 3.01



Querix 4GL

4GL Reference

Part Number: 019-022-301-014

Elena Krivtsova, Svitlana Ostnek
Technical Writers

Last Updated September 2014

Querix 4GL Reference

Copyright © 2006-2014 Querix Ltd. All rights reserved.

Part Number: 019-022-301-014

Published by:

Querix (UK) Limited. 50 The Avenue, Southampton, Hampshire,
SO17 1XQ, UK

Publication history:

December 2010:	Beta edition
July 2011:	First edition
July 2012:	Updated for Lycia II
April 2014:	Updated for Lycia 2.2

Last Updated:

September 2014

Documentation written by:

Elena Krivtsova, Svitlana Ostnek, Olga Gusarenko

Notices:

The information contained within this document is subject to change without notice. If you find any problems in the documentation please submit your comments by email to documentation@querix.com.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose without the express permission of Querix (UK) Ltd.

Other products or company names used within this document are for identification purposes only, and may be trademarks of their respective owners.



Table of Contents

INTRODUCTION	1
QUERIX 4GL LANGUAGE	2
CHAPTER 1: DATA TYPES	8
DATA TYPES OF QUERIX 4GL	9
ARRAY	16
BIGINT	18
BYTE	19
BOOL	20
BOOLEAN	21
CHAR	23
CHARACTER	24
CURSOR	25
DATE	26
DATETIME	27
DEC	32
DECIMAL (P,S)	33
DECIMAL(P)	34
DOUBLE PRECISION	35
DYNAMIC ARRAY	36
FLOAT	40
FORM	41
INT	42
INTEGER	43
INTERVAL	44
MONEY	50
NUMERIC	51
PREPARED	52
REAL	53
RECORD	54
SMALLFLOAT	57
SMALLINT	58
STRING	59
TEXT	60
TINYINT	61
VARCHAR	62
WEBSERVICE	63
WINDOW	64
CONVERSION OF DATA TYPES	65
DATA TYPES USED WITH DATABASES	70
CHAPTER 2: 4GL EXPRESSIONS	72
NOTION OF 4GL EXPRESSIONS	73
BOOLEAN EXPRESSIONS	81
INTEGER EXPRESSIONS	84
NUMERIC EXPRESSIONS	87
CHARACTER EXPRESSIONS	89
TIME EXPRESSIONS	91



FIELD CLAUSES	97
TABLE QUALIFIERS	99
ASTERISK (*) AND THRU/THROUGH KEYWORD	101
ATTRIBUTE CLAUSE	104
LOGICAL BLOCKS.....	110
CHAPTER CHAPTER 3: STATEMENTS	112
CALL	113
CASE.....	117
CLEAR.....	121
CLOSE DATABASE.....	123
CLOSE FILE.....	124
CLOSE FORM.....	125
CLOSE WINDOW.....	126
CONNECT TO	127
CONSTANT.....	131
CONSTRUCT	134
CONTINUE	152
CURRENT WINDOW.....	154
DATABASE.....	156
DEFER.....	160
DEFINE.....	162
DIALOG.....	170
DISCONNECT	190
DISPLAY.....	192
DISPLAY ARRAY.....	200
DISPLAY FORM	218
END	220
ERROR	222
EXIT	224
FINISH REPORT	227
FOR	228
FOREACH.....	231
FREE.....	236
FUNCTION	238
GLOBALS	244
GOTO	248
IF	250
IMPORT	252
INITIALIZE.....	256
INPUT	258
INPUT ARRAY.....	278
LABEL.....	304
LET	305
LOAD	308
LOCATE	313
MAIN	317
MENU	319
MESSAGE	333
NEED	335
OPEN FILE	336
OPEN FORM	339



OPEN WINDOW	342
OPTIONS.....	350
OUTPUT TO REPORT.....	361
PAUSE	363
PREPARE.....	364
PRINT.....	371
PROMPT.....	373
READ	382
REPORT	384
RETURN.....	388
RUN.....	391
SEEK.....	395
SET CONNECTION	396
SKIP.....	398
SLEEP.....	399
SQL.....	400
START REPORT	403
TERMINATE REPORT	411
TRY...CATCH.....	412
UNLOAD	415
VALIDATE	419
WHENEVER.....	420
WHILE	429
WRITE	431
CHAPTER 4: REPORTS	433
4GL REPORTS OVERVIEW	434
PRODUCING 4GL REPORTS	435
DEFINE SECTION.....	439
OUTPUT SECTION.....	441
ORDER BY SECTION	448
FORMAT SECTION	451
EXECUTABLE STATEMENTS IN THE REPORT PROGRAM BLOCK.....	459
EXIT REPORT.....	460
NEED	461
PAUSE	462
PRINT	464
SKIP.....	474
CHAPTER 5: PRE-PROCESSOR.....	475
PRE-PROCESSOR	476
APPENDIX A.....	484
4GL KEYS.....	485
APPENDIX B.....	490
CLASSES	491
INDEX	496





Introduction

This guide describes the features and syntax of the 4GL language, including 4GL data types, statements, reports, and operators. The description includes both the Informix 4GL features which are fully supported by the Querix 4GL language and the language features introduced by the Querix team. The additional features make the 4GL language more flexible and offer more resources for efficient application development.

This guide is written for 4GL developers. You need to be familiar with the general concepts of the 4GL language such as 4GL identifiers, program blocks statements, etc. However, you do not need the specific Informix 4GL language reference to use this manual, all the language features which are common for Informix 4GL and Querix 4GL are described here. The knowledge of SQL (structured query language), nevertheless, would be useful.

What Compiler Should Be Used

This manual was written with the assumption that you are using Lycia I or later. Lycia I and Lycia II have very much in common regarding the syntax of the statements and the data types, so practically everything written in this manual applies to both Lycia I and Lycia II.

You can still easily use applications developed with an earlier version of Querix 4GL or with the Informix 4GL of Version 7.2 and lower, with this version of 4GL.



Querix 4GL Language

Querix 4GL is a programming language that was developed on the basis of Informix-4GL language and is fully compatible with it. It introduces a large number of features which enhance Informix-4GL and widen the range of the possible uses of the language. Querix 4GL is similar to the English language, so it is easy to learn and use. It also includes the standard SQL language for accessing and manipulating a database.

Case Sensitivity

The source code in the source files as well as in the form files is not case sensitive. The capitalization of the keywords is done for easier code reading and is not obligatory.

In the literal character strings enclosed in single or double quotation marks the letter case is preserved. The character string can be used as character literals, file names, etc. If you work on Linux, you may want to preserve the letter case in the file names. In all other cases the strings that are not enclosed in quotation marks are switched to lower case during compilation. For example, in a variable name consisting of lower and upper case letters all the upper case letters will be turned to lower case letters during compilation.

Character Set

While Querix 4GL needs ASCII character set, data values, form specifications, reports and identifiers using client locale characters are also acceptable.

Special Symbols

The whitespaces, quotation marks escape symbols and delimiters used in Querix 4GL can have special usage.

Whitespaces

Extra number of white spaces between statements, statement elements, or operands in an expression is ignored. This way the whitespaces, the newline and tab symbols can be used to mark your code visually without any effect of its execution. Blank spaces cannot be used inside identifiers.

On the other hand white spaces can serve as delimiters in the following cases:

- as the delimiter between the command line arguments sent to the program
- if the whitespace is set as delimiter in the form file
- as delimiters in the unl files, if set accordingly

Quotation Marks

Single (') and double (") quotation marks are used to demark literal character strings in Querix 4GL. A character string should be marked with similar quotation marks at both sides:

```
DISPLAY "Correct quotation" AT 2,2
DISPLAY 'Another correct quotation' AT 3,2
DISPLAY "An incorrect quotation" AT 4,2
```

A quotation mark of a type that is different from that used at the beginning of the character string, is treated as a usual character. This allows to create quoted substrings :

```
DISPLAY "Type 'Q' to quit"
DISPLAY 'Type "Q" to quit'
```



If a database processes delimited SQL identifiers, the system reserves the (" ") marks to be used with SQL statements, therefore, it will be necessary to use only single quotation marks in most of the 4GL statements.

Backslash

The backslash symbol is a default escape symbol which makes the compiler treat the symbol following it as a literal character and ignore its special functionality. For example, the backslash (\) symbol can make the quotation mark be treated as a character instead of a string delimiter. In this case, a character string can include the quotation marks of the same type as those used to mark the character string itself:

```
DISPLAY "Type \"Q\" to quit"
```

Therefore, to add a backslash symbol to a character string, you have to use a double backslash in the source code; to add a double backslash, four symbols in a row are needed:

```
DISPLAY "This is a backslash: \\\"
```

```
DISPLAY "This is a double backslash: \\\\\\"
```

Backslash is also used to mark special symbols, such as a new line symbol: \n.

Semicolon

Semicolons are typically not needed in 4GL, except for some cases of PREPARE and PRINT statements usage as well as the END SQL keywords. However, the semicolon still can be used as a statement terminator.

Comments

Besides the source code, a 4GL source file can include comments - the text that is ignored during the compilation, is not displayed at runtime and does not influence the source code in any way. The comments may include notes, descriptions, annotations, explanations, i.e. any additional information which may help the person who will read or modify the source code in future.

Comment indicators

Comments may be placed at any part of the source code, but they should be identified by special symbols in order to make the compiler distinguish them from the meaningful source code. There are four types of comment indicators:

- The # ("pound", or "sharp") symbol indicates a one-line comment that starts right after this sign and continues only to the end of the line. Any symbols placed on this line to the right of this symbol are treated as comments. No closing symbol is required.
- Double hyphens or minus signs (--) works the same as the # symbol comment indicator.
- The double-brace symbols indicate a multiple-line comment. The comment begins right after the left-brace ({) and ends with the right-brace (}) closing symbol. These symbols can be placed on the same line or on different lines. All the symbols between the double-brace symbols become commented regardless of the number of lines.
- Double hyphens with an asterisk or a sharp symbol (--* or --#) is a comment indicator which is ignored by Querix 4GL compiler but recognized as a comment indicator by classical Informix tools. Therefore, these indicators can be used in applications, intended to be run by both Querix and third-party tools in cases, when a certain part of a code cannot be recognized or processed by the latter, but is runnable for Querix products.

Comments Restrictions

Several restrictions should be kept in mind when you create comments:

- Any comment indicator placed within a quoted string is treated as a part of this string, not as a comment indicator.



- Two minus signs in a row used in arithmetic expressions are will be treated as a comment indicator. To prevent this, the two consecutive minus signs should be separated with a blank space or parentheses (LET a = b -(-3), not LET a = b--3)
- The sharp symbol cannot be used to mark comments in prepared statements, form specifications and SQL statement blocks.
- It is impossible to nest comments within comments by using left- and right-brace symbols.
- The SCREEN section of a form-specification .per file cannot include comments.

Application structure

An application consists of modules - .4gl files with the source code, as well as additional files storing forms, messages, themes, etc.

Program modules and program blocks

The source code is organized in program blocks - units that consist of sets of 4GL statements which are intended to perform some part of the application functionality. There are three kinds of program blocks: MAIN, FUNCTION and REPORT. Each block should begin with a keyword, which corresponds to the name of the block, and end with the END keyword, followed by the block name (END MAIN, END FUNCTION, END REPORT).

Each application can include only one MAIN block, and as many FUNCTION and REPORT blocks, as it is needed.

The MAIN block is started automatically when the application launches; the FUNCTION and REPORT blocks are called by 4GL statements. C and ESQL/C functions are also supported by Querix compiler.

The application control is passed from one program block to another by means of CALL, RETURN and REPOERT statements as well as by 4GL expression referencing a programmer-defined function.

When creating a 4GL application, you should keep in mind the following rules:

- 4GL statements cannot be put outside a program block, except for some special cases (DATABASE, DEFINE, GLOBALS)
- Variables declared within one of the blocks have local scope of reference and cannot be referenced from outside their container block. For the details of variable scoping, see the "Variable Declaration" section of the DEFINE statement chapter of this guide.
- One program block cannot be placed in several modules at once.
- A control block cannot be nested within another one.
- GOTO or GO TO keywords can reference the labels placed only within the program block they occur.
- The DATABASE statement specified before the first program block, influences the compilation of the application. Placed within any of the program blocks, it is processed only at runtime.
- The effect of the WHENEVER statement starts when the application executes it and lasts till the next WHENEVER statement execution or to the end of the module where it is placed.

Compound statements

Some of the 4GL statements are compound statements, which means, that they can have other statements in their specification. Each compound statement begins with the statement keywords (INPUT ARRAY, IF, etc.) and ends with the END keyword followed by the statement identifier (END INPUT, END IF, etc.). Some of the statements can be used individually, without any internal statements (e.g. INPUT ARRAY statement without control blocks). In this case, the END keyword is not needed typically, except for some situations which are described in corresponding parts of this manual.

For most compound statements, EXIT *statement* keyword is also supported, which allows to terminate the statement execution under certain circumstances.



Below, is given the list of the compound statements supported by Querix 4GL:

- CASE
- CONSTRUCT
- DIALOG
- DISPLAY ARRAY
- FOR
- FOREACH
- FUNCTION
- GLOBALS
- IF
- INPUT
- INPUT ARRAY
- MAIN
- MENU
- PROMPT
- REPORT
- SQL
- TRY
- WHILE

Each compound statement can include one or several 4GL and/or SQL statements. In some cases, they need to be placed within specially defined control blocks, like ON KEY block, BEFORE ... and AFTER ... blocks for input statements, THEN and ELSE blocks for IF, etc. In other statements, like FOR or WHILE, special control blocks identifiers are not needed and the statements to be executed are placed just between the *statement* identifier and END *statement* keywords.

Querix 4GL allows some statement blocks like FUNCTION or REPORT blocks to be empty in order to enable the programmer to run applications at the development stage

Compound statements can nest and invoke other compound statements, except for some restrictions that are described separately.

4GL Identifiers

Some of the 4GL program elements have to be referenced by name in statements and specifications. The table below lists the most common application entities that need names, or identifiers, specification:

Program entity	Identifier specification
Functions and their arguments	FUNCTION statement
Program variables	DEFINE and GLOBAL statements
Reports and their arguments	REPORT statements
Screen arrays	ATTRIBUTE form file section or graphical form file properties section
Screen forms	OPEN FORM statement
Screen records	INSTRUCTIONS form file section or graphical form file properties section
Statement labels	LABEL statement
Table aliases	TABLES form file section or graphical form file properties section
Windows	OPEN WINDOW statement

Besides the entities given above, the names are needed for database objects and SQL entities.



4GL Identifiers Specification

A program entity identifier is a character string, created with the following rules being taken into considerations:

- The identifier should be from 1 to 128 characters long
- The identifier can include only ASCII letters, digits, and the underscore (_) symbol. Blank spaces, punctuation symbols and other symbols that do not represent alphabetic characters or numbers are not permitted.
- The identifier should begin with a character or an underscore symbol.
- Identifiers are not case sensitive
- 4GL identifiers should not match any of SQL, C and C++ keywords, otherwise, unexpected results can be obtained.

To use non-ASCII characters, you should set up the CLIENT_LOCALE so that they are specified in the locale code set. In Asian locales with non-alphabetic symbols (Chinese, Japanese, etc.) 4GL identifiers don't have to start with a letter, but their size should be no more than 128 bytes.

SQL Identifiers Specification

The rules of SQL identifiers specification are mostly similar to those of the 4GL identifiers and correspond to generally applied rules for SQL identifiers.

Querix 4GL allows SQL identifiers to include non-English characters, if they are supported by the system locale settings, the database and its connectivity tools.

SQL identifiers may match 4GL identifiers, but in this case their scopes of reference should be kept in mind carefully in order to prevent unexpected performance.

SQL Identifiers Visibility

By default, a prepared object or a cursor is visible from the DECLARE or PREPARE statement where it occurs to the respective FREE statement or to the end of the module where it is specified. The DECLARE and PREPARE statements, following the FREE statement in the same module, cannot be used to specify the same name as a previously freed object had.

Any other SQL identifier is of a global scope of reference.

Global database elements (index, column, table, etc.) cannot be referenced, if their database is not opened or is inaccessible. Neither can they be referenced after the application executed the DROP statement with their identifier.

If an application has matching 4GL and SQL identifiers referable in the same context, the 4GL gets the higher precedence. The @ symbol can be placed before the columns and tables names to distinguish them from 4GL identifiers

User Interaction

Querix 4GL supports the following interface elements that provide interacting with users:

- Menus
- Toolbars
- Screen forms
- 4GL windows
- Pop-up message and dialog windows
- Help messages
- Reports based on database data



All of these elements are briefly overviewed in this section and are described in details in the corresponding parts of this guide.



CHAPTER 1: DATA TYPES

THIS SECTION WILL LIST AND DESCRIBE ALL THE DATA TYPES AVAILABLE IN QUERIX 4GL IN ALPHABETICAL ORDER. IT WILL ALSO GIVE THE GENERAL OVERVIEW OF THE DATA TYPES AND GROUP THEM ACCORDING TO THE TYPES.



Data Types of Querix 4GL

This chapter will touch upon all the data types present in Querix 4GL, their usage and conversion. Querix 4GL data types include the following enhancements:

- New data types are introduced, they are BOOLEAN and STRING
- With the introduction of Lycia, all data types are now objects and the methods described at the end of this chapter can be applied to them.

Data Values

Data values in Querix 4GL must belong to some data type. Data type is a 4GL category that describes which type of data is stored in a variable or a column, etc. The types of operations that can be performed on a value depend on its data type.

Some data types are compatible; this means that a value of one data type can be converted to another data type. The process and conditions of conversion are described later in this guide.

Querix 4GL data types described in this chapter are listed in alphabetical order. The U.S. English locale is used for describing the display and input format of the data as the default.

Classification of Data Types

The following 4GL objects require a data type to be declared:

- A variable;
- A formonly field;
- A formal argument used in a function or a report (it cannot be of ARRAY data type)
- A value that is returned by a function

Below, there is the list of the data types that can be used for declaring the above listed objects together with the list of the corresponding values:

Data Type	Corresponding Values
ARRAY	Sets of values of any single 4GL data type. The upper bound for the number of array elements must be set
BOOLEAN/BOOL	Provides a simple Boolean type, variable of this data type can be either TRUE or FALSE
BIGINT	Stores 8-byte integer values in the range $-(2^{63} - 1)$ to $2^{63} - 1$
BYTE	Binary data of any kind, up to 2^{31} bytes long
CHAR/CHARACTER (size)	Character strings; (size) is the number of bytes used to store the value; it can be up to 32,767 bytes long. CHAR is a synonym for CHARACTER
CURSOR	An object that stores information about a cursor
DATE	Specific calendar dates
DATETIME	Specific calendar dates together with time values
DEC/DECIMAL(p,s)	Fixed-point numbers, with a specified precision and number of decimal places. DEC is a synonym for DECIMAL
DEC/DECIMAL(p)	Floating-point numbers, with a specified precision but with undefined number of decimal places. DEC is a synonym for DECIMAL
DOUBLE PRECISION /	Floating-point numbers with up to 14 significant digits.



FLOAT (p)	DOUBLE PRECISION is a synonym for FLOAT
DYNAMIC ARRAY	Sets of values of any single 4GL data type. It has no obligatory upper bound for the number of its elements
FORM	An object that stores information about a form
INT/INTEGER	Any whole numbers within the range -2,147,483,647 to 2,147,483,647. INT is a synonym of INTEGER
INTERVAL	A defined period of time; can be measured in hours and minutes or months and years
MONEY(p,s)	Units of currency, with definable precision and number of decimal places
NUMERIC	It is a synonym for DECIMAL
PREPARED	An object that stores information about a prepared statement
RECORD	An ordered set of values, which can be made up of any combination of data types
SMALLFLOAT/REAL(p)	Floating-point numbers, with up to 16 significant digits. REAL is a synonym for SMALLFLOAT
SMALLINT	Whole numbers within the range -32,767 to 32,767
STRING	A variant length holder for character strings.
TEXT	Character strings, up to 2^{31} bytes in length
TINYINT	
VARCHAR(size)	Character strings, the length of a string is limited only by the overall storage capacity of the system; (size) is the number of bytes used to store the value, it can be up to 225 bytes long
WEBSERVICE	An object which stores information about a webservice
WINDOW	An object which stores information about a 4GL window

All the data types except for the RECORD, ARRAY and DYNAMIC ARRAY data types correspond to the built-in SQL data types. The data types of Querix 4GL can be regarded as a set of the SQL data types that are recognized by Querix , tools thought the following restrictions are applied:

- The SQL data type SERIAL cannot be used in Querix 4GL statements, you should substitute INTEGER for SERIAL data type in the statements that are not SQL statements.
- NCHAR and NVARCHAR data types are recognized by Querix 4GL. However, they are mainly used to work with the database columns of VCHAR and NVARCHAR data types.
- Querix 4GL does not recognize the BITVARYING, BLOB, CLOB, DISTINCT, LIST, MULTISET, OPAQUE, REFERENCE, ROW, SET, and the user-defined data types of Informix database servers.

All Querix 4GL data types can be divided into four groups:

1. **Large** data types (TEXT and BYTE)
2. **Structured** data types (RECORD, ARRAY and DYNAMIC ARRAY)
3. **Object** data types (CURSOR, FORM, PREPARED, WEBSERVICES, WINDOW)
4. **Simple** data types (the rest of Querix 4GL data types)

Simple Data Types

A simple 4GL data type can store a single value. The maximum memory used for the storage of a value is set implicitly by the data type declaration or explicitly using indication of *size*, *scale* and *precision*.

Indicator	Description	Data Types
(size)	This is the largest amount of bytes that the data type	CHAR (32)



	can hold. For CHAR the range can be $1 \leq size \leq 32,767$. There is no upper limit for bytes (characters) in the VARCHAR data type, it is limited only by the free memory of the system; the lower limit is 1 as well as in the CHAR data type. The default number of characters is 1	VARCHAR (453)
precision (p)	This is the number significant of digits that can be used. The significant digits are any digits appearing within the number except for the leading zeros, e.g. number 123,45 has 5 significant digits and its precision is 5. For FLOAT the range is 1 to 14 significant digits. For DECIMAL & MONEY the range is 1 to 32. The default number of digits for DECIMAL and MONEY is 16.	FLOAT (3) DECIMAL (3) DECIMAL (7, s) MONEY(6, s)
scale (s)	This is the number of decimal places. There should be 32 or fewer decimal places and the number of decimal places should not exceed the number of significant digits ($scale \leq precision$). The default number of decimal places for MONEY is 2.	DECIMAL (7, 5) MONEY (6, 2)

All the above mentioned parameters should be specified using numbers from 1 to 9 and 0, without embedded blank spaces or commas, and without a decimal point.

The simple data types can be subdivided into three large groups: number, time and character.

Numeric Data Types

The numeric data types in their turn are subdivided into whole numbers (INTEGER, SMALLINT, BIGINT, TINYINT, BOOLEAN), floating-point numbers (DECIMAL(p), FLOAT, SMALLFLOAT) and fixed-point numbers (DECIMAL(p, s), MONEY):



	Data Type	Description
Whole Number	BIGINT	Integer, ranging from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 (that is, $-(2^{63}-1)$ to $(2^{63}-1)$)
	BOOLEAN/BOOL	Either 1 or 0
	INTEGER/INT	Integer numbers from -2,147,483,647 to 2,147,483,647
	SMALLINT	Integer numbers from -32,767 to 32,767
	TINYINT	Integer numbers from -128 to 127
Fixed-point	NUMERIC / DEC / DECIMAL (<i>p, s</i>)	Fixed-point numbers of scale <i>s</i> and precision <i>p</i> .
	MONEY (<i>p, s</i>)	Currency values, of scale <i>s</i> and precision <i>p</i>
Floating-point	NUMERIC / DEC / DECIMAL (<i>p</i>)	Floating-point numbers of precision <i>p</i>
	FLOAT / DOUBLE PRECISION	Floating-point numbers with double-precision
	SMALLFLOAT / REAL	Floating-point numbers with single-precision

In the table above *p* stands for precision, *s* stands for scale.

Querix 4GL also supports the data types of the IBM Informix Dynamic Server. These data types can be used only in SQL statements (ALTER TABLE, CREATE TABLE, etc.) embedded into 4GL source code. These data types can be used only to specify the data types of the table columns, to insert values into these columns you should use corresponding 4GL data types.

Data Type	Description	Corresponding 4GL Data Type
INT8/INTEGER8	Stores 8-byte integer values in range $-(2^{63}-1)$ to $2^{63}-1$	BIGNIT
SERIAL	Stores large sequential integers in same range as INT	INT/INTEGER
SERIAL8	Stores large sequential integers in same range as INT8	BIGINT

Character Data Type

Querix 4GL supports two character data types:

Data Type	Description
CHAR/CHARACTER (<i>size</i>)	Strings with the <i>size</i> of up to 32,767 bytes
STRING	Character strings without the defined size. The size of a STRING variable depends on the size of the value stored in it. The maximum size is not limited by 4GL; it is only limited by the storage capacity of the system.
VARCHAR (<i>size</i>)	Strings with defined <i>size</i> . The <i>size</i> is not limited by 4GL; it is only limited by the storage capacity of the system. This data



type reserves free memory space equal to *size* for the variable which has been declared as VARCHAR data type.

The TEXT data type is not included into Querix 4GL character data types. This data type can store text strings of up to two gigabytes, but TEXT data type is manipulated in a way that differs from the way the CHAR, STRING and VARCHAR data types are manipulated.

Time Data Types

Querix 4GL supports three simple data types that can be used for time indication. Two of them store specific moments in time, and the third stores periods of time which can be either positive or negative.

Data Type	Description
DATE	Calendar dates, including day, month & year values for any day between 1 January, 0001 and 31 December, 9999.
DATETIME	The same as DATE, but with additional divisions for hours, minutes, seconds and fractions of seconds.
INTERVAL	Spans of time, measurable in units from years to months and from days to fractions of seconds.

Structured Data Types

Structured data types are used to store a number of values as a structured set. There are three structured data types in Querix 4GL:

Data Type	Description
ARRAY	An ordered three-dimensional set of values (can have up to 32,767 values in each dimension) that belong to any single data, provided that all the values are of the same data type
DYNAMIC ARRAY	Arrays of values (in up to three dimensions) that belong to any single data type. The optional upper bound should be greater than 0 and less than or equal to 2,147,483,647.
RECORD	An ordered set of values of any data type. Values within one RECORD can be of different data types

Object Data Types

The new types of data have been introduced to make the static objects dynamic. Such objects as screen forms now can be manipulated like other data types. They can be declared like variables of simple data types, thus they can obtain any scope of reference depending on the declaration context. They can also be passed to functions and returned by them.

Data Type	Description
CURSOR	Contains a cursor identifier and the statement for which it is declared
FORM	Contains a form name and the name of the form file
PREPARED	Contains an identifier of a prepared statement and its text
WEBSERVICE	Contains the information about a web service
WINDOW	Contains a window identifier, its position, size and attributes



Large Data Types

Large data types store links to binary objects which size can be up to two gigabytes. The size of such values can be smaller, if your system imposes some limitations.

Data Type	Description
TEXT	Strings of printable characters that may include whitespace characters
BYTE	Anything that can be stored in a digital form (including images, sounds, programs, etc.)





ARRAY

ARRAY is a structured data type that stores a one-, two-, or three-dimensional array of variables of identical data types. The size of each dimension is a positive integer of up to 32,767. Dimensions can be of different size and they can contain variables of any data type, provided that all the elements of the array are of the same data type.



Element	Description
Size	The upper bounds for up to 3 dimensions. Can include from 1 to 3 4GL expressions separated by commas that return positive integers
4GL Data Type	Any Querix 4GL or user-defined data type

A variable included into an ARRAY is called an *element*. The example below declares a three-dimensional program array called *my_array*:

```
DEFINE my_array ARRAY[a,b,c] OF INTEGER
```

- **a** indicates a-th element of a single-dimensional array. E.g. *my_array[3]* represents a variable on the third row of the array.
- **b** indicates b-th element in the a-th row within an array that contains two dimensions. E.g. *my_array[3, 5]* represents the fifth element in the third row of the array
- **c** indicates the c-th element in the b-th column of the a-th row within an array with three dimensions. E.g. *my_array[3, 5, 1]* represents the first element in the fifth column of the third row of the array.

	Note: In the SQL...END SQL statement, the variable that represents an array needs a dollar sign (\$) preceding it. However no dollar sign is required for a variable used as a coordinate in an array
--	--

Here are several examples of an array declaration:

```
DEFINE array1 ARRAY[100,100,100] OF RECORD rec1 LIKE client.*  
DEFINE array4 ARRAY[500] OF RECORD  
    variable1 LIKE client.fname,  
    variable2 INT  
END RECORD  
DEFINE array2 ARRAY[7,100] OF VARCHAR(20)  
  
DEFINE array3 ARRAY[1000] OF my_rec
```

In the last example *my_rec* is a user-defined type of record (created with the help of DEFINE...TYPE AS statement).



Individual Array Elements

You cannot manipulate an array as a single unit; you can only operate its individual elements. A single element can be manipulated by specifying its co-ordinates next to the name of the array in which it is included. The number of co-ordinates an element requires depends on how many dimensions there are in the array. A coordinate must be specified for each dimension of a multi-dimensional array. If a co-ordinate for one or two dimensions is missing, though it is required, 4GL will produce a compile-time error.

You can operate an array as a unity, so it can be passed to and from a function. They are passed by reference.



Note: It is also possible to pass a record that contains an array or a dynamic array as its member as a single unit. However, make sure you avoid cyclic references of such records.

Substrings of Array Elements of Character Type

A substring of an array is a list of character values within an individual element of an array of character type (CHAR, STRING or VARCHAR). You can use a pair of integer expressions between square bracket ([]) to specify a character substring within the string value of the array element. E.g. If my_array[d, e, f] is an element of a three-dimensional array of character data type, you can specify a substring within this element as follows: my_array[d, e, f] [m, n] where *m* and *n* specify the positions of the first and the last character (respectively) of the substring within the array element my_array[d, e, f]. The *m* and *n* should be positive integers, *m* should be smaller than *n*.



BIGINT

This data type is used to store 8-byte whole numbers whereas INTEGER data type can store only 4-byte numbers. The BIGINT data type can store values in a range from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807.

Thus the BIGINT data type is typically used to store values which exceed the limits of INTEGER values. It can also be used to store values from the database columns of INT8 or SERIAL8 data types. The BIGINT data type can pass values to C functions, where they can be converted into a 8-byte C whole-number data type.



BYTE

Any digital data can be stored in a BYTE data type. The BYTE data type stores information unstructured, these can be program modules, pictures, sounds and any other information that can be stored in digital form.

You can use the LIKE keyword within the DEFINE statement to define a variable like a database column of the BYTE data type.

The value of BYTE type is limited to 2 gigabytes, but it can also be limited by the storage capacity of your system.

A variable of this data type can be used:

- In operations with database columns of BYTE data type
- To reference a file you need to use in your program

After a variable of BYTE data type has been declared, it is necessary to use LOCATE statement to tell the program, where the information for this variable is located. A column defined as BYTE type can be assigned to a BYTE variable completely or partially. If you want to assign only a part of the contents of the column to a BYTE variable you'll need to specify the beginning and the end of the part you want to assign in bytes using square brackets as delimiters – *variable [m, n]*. E.g.:

```
SELECT image1 [1,100] INTO my_pic FROM catalogue
```

Here *m* and *n* are the coordinates of the first byte of a picture from which the part you want to assign begins and the last byte of the piece. The above code will select first 100 bytes of the specified picture *image1* into *my_pic* variable.

Restrictions applied to BYTE Data Type

There are a number of restrictions that should be kept in mind while using BYTE data type:

- Only IS NULL and NOT NULL operators can be used with variables of this type.
- The following statements cannot be used with BYTE values: DISPLAY, PRINT, LET, INITIALIZE. (The later two statements cannot assign a value to a BYTE variable other than NULL)
- BYTE arguments can be passed by reference using CALL and OUTPUT TO REPORT statements, but they cannot be passed by value.
- A form field assigned to a database column of BYTE data type or a formonly field of BYTE data type will display the BYTE value, if this value is an image of any type - a so called BLOB. If the BYTE value is some data which cannot be represented graphically, the BYTE field will remain empty.
- A BYTE value can be edited by means of the PROGRAM attribute.
- A form field of BYTE type can have no other attributes except COLOR and PROGRAM



BOOL

It is a synonym of the BOOLEAN data type.



BOOLEAN

The BOOLEAN data type provides a simple Boolean type. A variable or the BOOLEAN data type can acquire the following values:

- 1, which stands for TRUE
- 0, which stands for FALSE
- NULL

This data type can be used in Boolean expressions and in other cases when either true or false result is required.

The values can be assigned to such kind of value in three different ways:

Method	Example	Restrictions
Assigning a whole number	LET bool_var = 1 LET bool_var = 0	You cannot assign any other numbers either whole or fractional except 1 and 0
Assigning specific keywords	LET bool_var = NULL LET bool_var = FALSE	The only keywords that can be assigned to a Boolean variable are TRUE, FALSE, and NULL
Assigning values during input		You can print either 1 or 0 for the BOOLEAN value, no other characters are allowed

The value of a BOOLEAN variable is stored in the form of integer either 1 or 0. However, if this variable is used in a comparison or in any 4GL expression, you can use either of the ways described above. e.g.:

```
DEFINE bool_v BOOLEAN
...
IF bool_v = TRUE THEN
...
END IF
IF bool_v = 0 THEN
...
END IF
```

Conversion of the BOOLEAN Data Type

The BOOLEAN data type has a wide range of conversion possibilities. All simple Querix 4GL data types can be converted into the BOOLEAN data type and the BOOLEAN data type can be converted to most of the simple 4GL data types except for the Time data types. The summary table for the data type conversions can be found in the "[Summary on Simple Data Type Conversion](#)" below in this chapter.

Values of BOOLEAN data type can be converted to simple Querix 4GL data types using the following rules:

Source Data Type	Resulting Data Type	Restrictions	Resulting Values		
			TRUE	FALSE	NULL
BOOLEAN	CHAR, VARCHAR, STRING	no restrictions	"1"	"0"	NULL
	INT, SMALLINT, DECIMAL, FLOAT, SMALLFLOAT	no restrictions	1	0	NULL



MONEY	no restrictions	\$1.00	\$0.00	NULL
DATE, DATETIME, INTERVAL	cannot be converted	-	-	-

Values of simple 4GL data types can be converted to the BOOLEAN data types using the following rules. The values are converted into TRUE, FALSE or NULL depending on the contents:

Source Data Type	Resulting Data Type	Restrictions	Values converted into:		
			TRUE	FALSE	NULL
CHAR, VARCHAR, STRING	BOOLEAN	no restrictions	"1", "TRUE", "true"	"0", "FALSE", "false"	Any other value or NULL
INT, SMALLINT, DECIMAL, FLOAT, SMALLFLOAT, MONEY		no restrictions	non-zero values	zero values	NULL
DATE, DATETIME, INTERVAL		no restrictions	non-zero values	zero values	NULL



CHAR

A CHAR data type stores a character string, the length of that string is limited by the maximum *size* specified when the data type is declared: CHAR (*size*). The character string can consist of printable characters and whitespaces.

The minimum size of the CHAR value is 1 byte; it is also the default maximum size of a CHAR variable which is applied if no *size* is defined explicitly. The maximum size that can be assigned to a CHAR value is 32,767 bytes. A variable defined as CHAR(53) can contain 53 or fewer bytes, but it cannot contain more than 53 bytes.

The length of a formonly field of the CHAR type cannot be specified, its default size corresponds to the field length from the screen layout.

When a value is passed between 2 variables of the CHAR type or between a variable and a column, both of which are of CHAR type, the number of bytes defined by the *size* option is transferred. That means that if a value is shorter than the size declared, additional whitespaces will be added to its end to fit the declared size. If the value entered for a variable or a column is longer than the *size* declared, it will be truncated on the right.



Note: Two variables of the CHAR data type can sometimes cause a comparison to fail because the empty spaces may be added to their ends. The number of empty spaces in them may differ because they have been declared with different *sizes*, though the visible characters are the same. You may use CLIPPED operator to solve this problem.

Every value of the CHAR data type has an ASCII 0 end-of-data character at the end of it. It often results in impossibility of subsequent data addition or retrieval to or from a CHAR database column. Use the CLIPPED operator to convert CHAR variables which have NULL values to empty strings.

CHAR variables can store any printable symbols including digits, but if you want to perform arithmetic operations on numbers, you should store these numbers in variables of numeric data types. The CHAR data type is generally compatible with number data types, but you might not be able to use them in some calculations.

There are situations when CHAR data type is preferable for storing number values. E.g. some postal codes can contain leading zeros, which are removed from values as insignificant, if they are stored as some of the numeric data types (INTEGER, SMALLINT). In such cases numbers should be stored as CHAR data type.



Note: The CHAR data type usually requires 1 byte per character so the size in bytes is equal to the size in characters, but in some East Asian locales a character may require more than one byte to be stored. Some white space characters can also occupy more than one byte.



CHARACTER

It is a synonym of the CHAR data type.



CURSOR

The CURSOR data type is an object data type which stores the cursor identifier. A cursor can be declared implicitly without using the CURSOR data type - by means of the DECLARE statement. Such cursor has global scope of reference. Using the CURSOR data type to declare a cursor explicitly a programmer can specify any desired scope of reference in the same way like for any other variable. Thus a CURSOR variable defined in the GLOBALS file will have the global scope of reference and a variable defined within a function will have the local scope of reference.

CURSOR variables can be passed to functions and returned by them in the same way as the variables of simple data types. You should not assign values to variables of this data type using the LET statement, for it may cause unexpected behaviour.

Both explicitly and implicitly declared cursors are mangled internally to ensure compliance with name size restrictions, and to ensure uniqueness. Use the CURSOR_NAME() function to retrieve the name of a cursor declared as a data type with the CURSOR variable as its argument.

A CURSOR variable can be used with the special methods which declare the cursor, open it, fetch the rows, etc. These methods are different depending on the type of the cursor (select or insert). The methods are called in the same way as other functions and are used together with the CURSOR variable in the following manner:

```
variable.Method()
```

The methods used with the CURSOR variable to create and process a select cursor are as follows:

- Declare() - declares the cursor
- SetParameters() - sets the values for the placeholders, if any
- Open() - opens the cursor
- SetReturns() - specifies variables to receive the returned records
- A number of methods performing fetching, e.g.: Fetch(), FetchNext(), FetchFirst(), etc..
- Close() - closes the cursor

An insert cursor needs a transaction to be opened first and then it will be typically used with the following methods:

5. Declare() - declares the cursor
- Open() - opens the cursor
- SetParameters() - specifies the values for the placeholders, if any
- Put() - sends the rows into the buffer
- Flush() - inserts the rows into the table
- Close() - inserts the rows into the table and closes the cursor

These methods are described in details in the "Lycia Functions Guide".



DATE

Calendar dates are stored using the DATE data type. The value of DATE type is stored as an integer that represents the number of days that have passed since December 31, 1899. If a DATE value represents a date before December 31, 1899, the stored integer is negative. The default display format of a DATE value depends on the settings of the system on which it is used. If the U.S. English settings are used as default for the system, the default format of a DATE value is:

mm/dd/yyyy

where *mm* stands for a month (acceptable values are 1 to 12), *dd* stands for a day (acceptable values are 1 to 31) and *yyyy* stands for a year (acceptable values are 0001 to 9999).

```
LET my_birthday = "11/28/1978"
```

The default format can be changed with the help of DBDATE environment variable which can change the delimiter symbol or the order of units.

The GL_DATE environment variable can specify a wider range of formats than the DBDATE variable can. E.g. In some East Asian locales it can make DATE data type accept and display Taiwanese or Japanese eras.

If a user enters one or two symbols for the value of a year (5 or 05), 4GL uses the settings of the DBCENTURY environment variable to restore the first two or three digits of the year. If you want to specify a year outside the current century, you must specify all the four digits, e.g. 1805.

	Note: If you want to specify a year of the first century Anno Domini (which is also known as Common Era - CE), you should enter leading zeros, e.g. 053 or 0053 for the year 53 A.D. It is not possible to use DATE data type to specify years BC.
--	---

The settings of the DBCENTURY variable do not affect dates stored in CHAR, STRING and VARCHAR data types. To override the settings of this variable for the DATE data type, the individual field attribute CENTURY may be used for the form fields of DATE type.

You can perform arithmetic operations using variables of DATE type, because they are stored as integers. E.g. you can use two values of DATE data type in an arithmetical expression to find the difference between them. You will get a positive or negative value of the INTEGER data type that will represent the number of days that have passed between the two dates. This integer value can be converted to INTERVAL data type with the help of UNITS DAY operator. However, you will not get meaningful values if you try to divide or multiply values of DATE type.

You can change the default display of the DATE data type in screen forms using FORMAT attribute. The DATE data type can accept the following formats for months: 01, 1, January. It can accept 1 or 01 for the day format.



DATETIME

The DATETIME data type stores the data about the specified moment in time, as well as the DATE data type, but it includes also the time of day values besides the date. You can specify the precision of the date and time value to be used: the range may vary from year through a fraction of a second. The DATETIME values are stored as fixed-point DECIMAL numbers.

The DATETIME data type has the following structure, which is used in the data type definition:



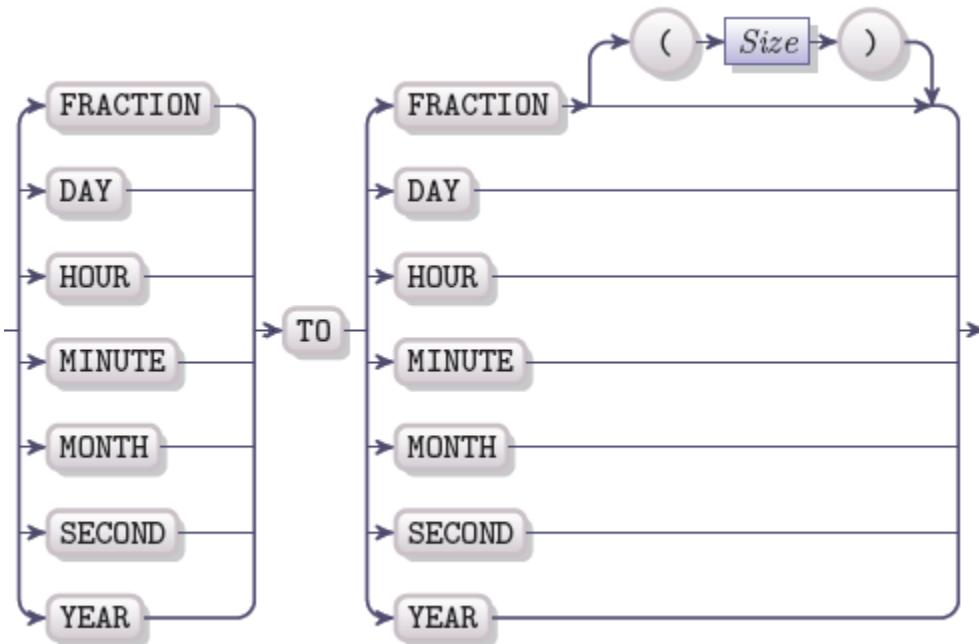
Element	Description
DATETIME Qualifier	Specifies the precision and scale of a DATETIME value.

The system environment variable GL_DATETIME can specify a wider range of formats than the DBDATETIME variable can. E.g. In some East Asian locales, it can make DATETIME data type accept and display Taiwanese or Japanese eras. In other locales it can also specify non-default display formats for DATETIME values.



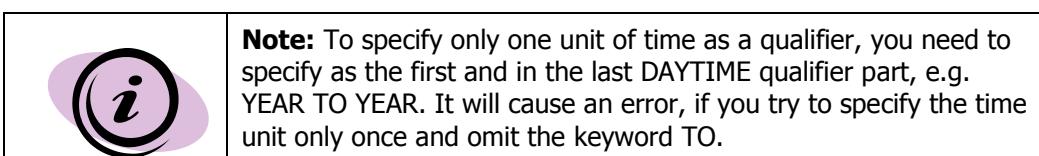
The DATETIME Qualifier

The first part of the DATETIME qualifier contains the information about the largest unit of time used in the DATETIME value and the second part contains the information about the smallest one. The DATETIME qualifier has the following structure:



Element	Description
YEAR	Specifies that years are used as the largest or the smallest time unit
MONTH	Specifies that months are used as the largest or the smallest time unit
DAY	Specifies that days are used as the largest or the smallest time unit
HOUR	Specifies that hours are used as the largest or the smallest time unit
MINUTE	Specifies that minutes are used as the largest or the smallest time unit
SECOND	Specifies that seconds are used as the largest or the smallest time unit
FRACTION	Specifies that fractions of a second are used as the largest or the smallest time unit
Size	An integer from 1 to 9 (in case of FRACTION from 1 to 5) that specifies the precision of the DATETIME qualifier

The time unit in the last qualifier may be of the same size as the time unit in the first one, but it cannot be a larger time unit.





The following are valid examples of qualifier usage:

```
DEFINE  
    dt_1 DATETIME YEAR TO HOUR  
    dt_2 DATETIME MONTH TO FRACTION
```

This is an **invalid** example of the qualifier usage which will produce an error when compiled:

```
DEFINE  
    d_time DATETIME HOUR TO YEAR
```

If a qualifier begins with MONTH and omits the year, 4GL uses the system date to supply any additional precision.



Note: If you use a qualifier that begins with DAY and omits MONTH, you may receive unexpected results, in case the current month defined by system clock-calendar has fewer than 31 days. It is advisable not to use qualifiers with the DAY as the first part of the qualifier.

The DAYTIME literal

There are two ways in which 4GL can assign value to a DATETIME variable:

- As a DATETIME literal
- As a character string

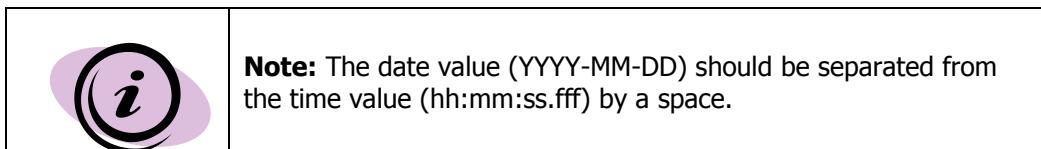
In the DATETIME literal, the DATETIME keyword is followed by whole numbers enclosed in parentheses that represent date and time values:



Time Unit	Description
Numeric Date and Time	A numeric DATETIME value entered in compliance with the format described below
Day Time Qualifier	The scale and precision specification that matches the assigned value
YYYY	A numeric indication of a year represented by integers from 0001 to 9999
MM	A numeric indication of a month represented by an integer from 1 to 12
DD	A numeric indication of a day represented by an integer from 01 to 31
- (Hyphen)	A delimiter used between year and month and between month and day
Space	A space is required between year-month-day sequence and time sequence
hh	A numeric indication of hours represented by an integer from 0 to 23



mm	A numeric indication of minutes represented by an integer from 0 to 59
ss	A numeric indication of seconds represented by an integer from 0 to 59
: (Colon)	A delimiter used between hours and minutes and between minutes and seconds
fff	A numeric indication of the fractions of a second: up to five digits
. (Dot)	A delimiter which separates fractions from seconds



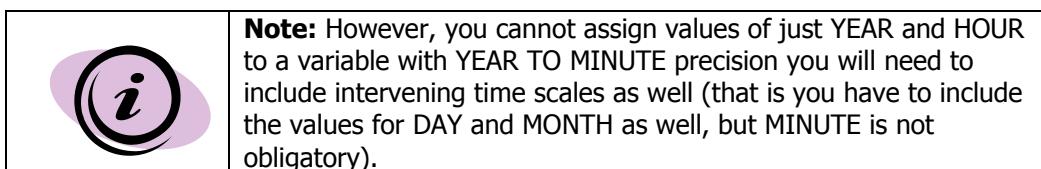
The default precision for the year is 4, month, hours, minutes and seconds have the default precision of 2 and this cannot be changed. In the source code, this diagram may be represented as follows:

```
LET my_dt = DATETIME (10-03-15 21:05:49.032) YEAR TO FRACTION (3)
```

The value "2010-03-15 21:05:49.032" will be assigned to the variable *my_datetime*, though only 2 digits have been entered for YEAR value, 4GL retrieves the first two digits from the system clock-calendar.

As well as in DATE data type you must include leading zeros to the year number, if you want to specify a year of the First Century (E.g. 0067 for 67 A.D.). If a user enters one or two symbols for a year value (1 or 01), they will not be regarded as a year of the First Century; 4GL uses the settings of the DBCENTURY environment variable to restore the first two or three digits of the year. To override the settings of this variable for the DATETIME data type, the individual field attribute CENTURY may be used for the form fields of DATETIME type.

It is not necessary to specify every time unit from the declared data type, you can specify only units you need. E.g. if you have declared a DATETIME variable with YEAR TO MINUTE precision, any values can be assigned to it provided that they do not exceed the declared precision. This means you can assign a MONTH TO HOUR, MONTH TO MINUTE, YEAR TO MINUTE, YEAR TO MONTH or YEAR TO HOUR values to it, but values that include SECOND and FRACTION will not be valid for a variable declared with YEAR TO MINUTE precision. You need to specify only those time units in the numeric value which correspond to the qualifier you use within the DATETIME literal. You cannot use them in random order or miss the successive time units between the largest and the smallest time unit you use.



If not all of the time units that have been declared for the variable are specified while assigning a value, 4GL expands the DATETIME value automatically to fill the time units to which no values were assigned. The missing values larger than the largest value assigned to a variable are retrieved from the system clock-calendar. If the omitted units are smaller than the smallest assigned value they are set to zero (MONTH and DAY are set to 1).



The Numeric Date and Time Value

A DATETIME value can be also specified as a character string which is called the numeric datetime value. It must be used without the DATETIME keyword and it must be enclosed in double quotation marks ("") or in single quotation marks (''). All the delimiters must still be used as described for DATETIME literals. A character string must include values for all the time units declared in the DATETIME qualifier. For a variable *cur_datetime* declared as DATETIME YEAR TO MINUTE the following statement will be valid:

```
LET cur_datetime = "2010-01-31 09:35"  
LET cur_datetime = '2010-01-31 09:35'
```

An error will occur, if you use the following **invalid** statements for *cur_datetime* variable:

```
LET cur_datetime = "2010-01 09:35"  
LET cur_datetime = '2010-01-31 09:35:20.02'
```

An error will also occur, if additional blank spaces are included into the string.

When values are entered into a form field of DATETIME type or during a PROMPT statement for a DATETIME variable the only valid format is an unquoted numeric DATETIME value. It must be a continuous sequence of values for all the time units declared or a part of it ordered continuously with the corresponding delimiters.

DATETIME values entered into form fields do not necessarily need to include all the time units that have been declared. It is not possible to use a DATETIME literal for form fields, here character strings are to be used instead. DATETIME literals are used in 4GL statements and in the data type declarations of formonly DATETIME fields.

By default, all time units of the DATETIME data type are represented by two-digit numbers except for the values of year and fraction. A fraction can contain up to 5 digits and is rounded up to an even number, it consists of 3 digits by default. The following formula can be used to calculate the number of bytes required to store a DATETIME value:

$$\frac{\text{total number of digits of all the time units}}{2} + 1$$

E.g. a YEAR TO DAY qualifier requires a total of 8 digits (4 for a year, 2 for a month, and 2 for a day). If we use the formula we will get: $((8/2) + 1) = 5$ bytes of storage.



DEC

It is a synonym for the DECIMAL data type.



DECIMAL (p,s)

The DECIMAL(*p,s*) data type is used to store fixed-point decimal numbers. This data type can contain up to 32 significant digits. The specification of precision (*p*) and decimal places (*s*) is optional. E.g. DECIMAL(10,2) indicates that a value may contain up to 10 significant digits and 2 decimal places to the right of the decimal point. The significant digits are all the digits present within the number except the leading zeros which are on the left from the decimal point. The significant digits include the digits both to the right and to the left of the decimal point. Thus number 1.567 will have 4 significant digits one of which is also the scale.

You should be aware that if a number fits the declared precision but the number of decimal places exceeds the scale declared, the value written to a variable will be rounded. E.g. if number 1.0004 is written to a variable of DECIMAL(5,2) data type, the value assigned to the variable will be "1.00". Thus the value has less than 5 significant digits, because it is limited by the number of available decimal places. Number 1.009 will be displayed as "1.01", because it will be rounded up.

The largest absolute value that can be stored in the DECIMAL(*p,s*) data type is calculated using the formula: $10^{p-s} - 10^s$ (where '*p*' is the precision and '*s*' is the scale).

Values that are less than $0,5 \times 10^{-s}$ are stored as zeros (where '*s*' is the scale).

It is impossible to specify precision or the number of decimal places for a formonly field of DECIMAL type. Its precision is always smaller than 32 and is equal to (*length of form field* – 2).

The DECIMAL data type with fixed point is usually used for storing data which require a set number of decimal places, e.g. rates or percentages and which require high precision.

Querix 4GL converts values entered for SMALLFLOAT and FLOAT data types from decimal numeration to binary numeration for storage purpose. To display data stored in these data types, Querix 4GL converts them back to decimal numeration format. Conversions may cause inaccuracy. It is advisable to use DECIMAL data type to store information for which high precision is desired.

The number stored as DECIMAL data type can have up to 32 digits to the left of the decimal point and up to 30 significant decimal digits on the right of the decimal point. The number of bytes required to store a value of DECIMAL data type is calculated using the following formulas.

$$\frac{\text{precision} + 3}{2}$$

When the scale is even:

$$\frac{\text{precision} + 4}{2}$$

When the scale is odd:

E.g. DECIMAL(10,2) requires ((10 + 3) / 2)=6 bytes of storage.



DECIMAL(*p*)

A $\text{DECIMAL}(p,s)$ value with specified precision and scale can be manipulated by the 4GL as a value with a fixed-point. However, precision and scale are optional, if the DECIMAL data type is declared without scale and precision, it is regarded as a floating-point number with 16 significant digits which is the default precision. The significant digits are all the digits present within the number except the leading zeros which are on the left from the decimal point. The significant digits include the digits both to the right and to the left of the decimal point.

If you specify only one parameter, it is interpreted as a precision and the number will be manipulated as a floating point number with the defined precision which may be different from the default one. The exponent of such number can range from 10^{-130} to 10^{126} . The absolute values can range from 1.0E-130 to 9.99E+126.

In an ANSI database, a column of a $\text{DECIMAL}(p)$ data type will store fixed-point numbers with the precision of *p* and a fixed zero scale. This number will be equivalent to an integer. However, if you declare a $\text{DECIMAL}(p, 0)$ data type the digits to the right of the decimal point will not be discarded and will be stored internally, though they will not be displayed. If you store a data value of 1.1 in the variable of $\text{DECIMAL}(5, 0)$ then this variable will be displayed as 1, but in Boolean expressions it will be greater than 1.



DOUBLE PRECISION

The DOUBLE PRECISION is a synonym for FLOAT data type.



DYNAMIC ARRAY

The DYNAMIC ARRAY data type works the same way as ARRAY data type, but it has some additional features. Use the following syntax to declare a **single-dimensional dynamic array** of variables:



Element	Description
4GL Data Type	Any Querix 4GL or user-defined data type except ARRAY, DYNAMIC ARRAY

Dynamic arrays differ from traditional 4GL program arrays in that they do not have a fixed size. Attempts to write beyond the bounds of a static program array will cause the array to resize automatically, whereas it is impossible to write beyond the bounds of a static program array.

A **multi-dimensional dynamic array** is declared using the following syntax:



Element	Description
4GL Data Type	Any Querix 4GL or user-defined data type except ARRAY, DYNAMIC ARRAY
Integer Expression	A 4GL expression returning a positive integer or a literal integer in the range from 1 to 3 which denotes the number of array dimensions

There exists a number of restrictions imposed on multi-dimensional dynamic arrays. They are as follows:

1. .getLength() method is not supported. E.g., for:

```
DEFINE a DYNAMIC ARRAY WITH 2 DIMENTIONS OF datatype
```

valid usage:

```
a[index].getLength()
```

invalid usage, resulting in a compile time error:

```
a.getLength()
```

2. DISPLAY ARRAY statement can be used only for the arrays of the RECORD data type display.

3. The same limitations as for the static arrays:

- definition: array length must always be greater than '0', otherwise an application fails to compile;



- verification of indices it bounds: on an attempt to access by the index falling outside beyond the range, a compile time error occurs

Usage examples:

1. A two-dimensional dynamic array declaration:

```
DEFINE x DYNAMIC ARRAY WITH 2 DIMENSIONS OF INTEGER
...
FOR i = 1 TO 100
FOR j = 1 TO 100
    LET x[i,j] = i*j MOD 7
END FOR
END FOR
```

2. A three-dimensional dynamic array declaration:

```
DEFINE x DYNAMIC ARRAY WITH 3 DIMENSIONS OF INTEGER
...
FOR i = 1 TO 10
FOR j = 1 TO 10
FOR k = 1 TO 10
    LET x[i,j,k] = i*j*k MOD 13
END FOR
END FOR
END FOR
```

Array Size

A dynamic array is declared without the upper bound. The array is automatically resized whenever a value is added to it or deleted from it. The size of the dynamic array corresponds to the number of the currently occupied records. The example below will resize the array up to 1000 elements:

```
DEFINE x DYNAMIC ARRAY OF INTEGER
...
FOR i = 1 TO 1000 -- makes the array up to 1000 elements
    LET x[i] = i MOD 3
END FOR
FOR i = 101 to 1000 STEP -- the array is still 1000 elements, but
    -- 900 elements contain NULL values
    INITIALIZE x[i] TO NULL
END FOR
```



To make the array size smaller, you need to use delete() or resize() methods, because the array size only increases automatically, but it does not decrease.

As you can see, though a dynamic array does not have a declared size, you still can use [] operator for indexing the array. You can reference any populated record within the array using this operator. Unlike the static arrays, with the dynamic array you can use an index which is outside the current array size boundaries. The index not necessarily should follow the last populated row index, e.g. your array can contain 10 members, but you can use 100 as an index.

If you use an index outside the array boundaries for retrieving a value, a runtime error will occur. However, if you use the index to assign a value to an array element outside the boundaries, the value will be successfully assigned. The total array size is calculated as the largest index regardless of whether the elements in between the first and the last one are populated. Thus if you assign some values to 1st, 2nd and 100th elements only, the total array size will be 100 elements, even though the rest of the elements will contain default values according to the array data type (e.g. for the INTEGER data type the default value is 0).

```
DEFINE x DYNAMIC ARRAY OF INTEGER
...
FOR i = 1 TO 5 -- this array will be 5 elements in size
    LET x[i] = i
END FOR
DISPLAY x[2] -- will display 2
DISPLAY x[6] -- will cause a runtime error

INITIALIZE x[4] TO NULL
LET x[10] = 10 -- expands the array to 10 elements

DISPLAY x[4] -- will display nothing (was set to NULL)
DISPLAY x[10] -- will display 10
DISPLAY x[9] -- will display 0 as the default for INTEGER
DISPLAY x[11] -- will cause a runtime error
```

Features of Dynamic Arrays

In their declaration syntax, DYNAMIC ARRAY variables do not differ much from ARRAY variables. The usage of the two array types is practically the same which means that dynamic arrays of Querix 4GL differ much from the dynamic arrays of Informix 4GL.

Querix 4GL DYNAMIC ARRAY has the following features:

- DYNAMIC ARRAY can be of TEXT and BYTE data types.
- DYNAMIC ARRAY data type is not limited to 32,767 elements, the upper limit of elements is equal to 2,147,483,647 (which is the maximum limit for the INTEGER data type).
- DYNAMIC ARRAY variables can be passed to or returned from functions. They are passed by reference rather than by value.
- DYNAMIC ARRAY variables can be passed to reports.



- DYNAMIC ARRAY variables can be used in the INPUT ARRAY, DISPLAY ARRAY statements.
- DYNAMIC ARRAY variables can be used in an SQL...END SQL statement.

	Note: It is also possible to pass a record that contains an array or a dynamic array as its member as a single unit. However, make sure you avoid cyclic references of such records.
---	---

Using Dynamic Arrays

Dynamic arrays can be used in the DISPLAY ARRAY and INPUT ARRAY statements like static arrays. When used in the INPUT ARRAY statement, the array will be automatically resized when the cursor moves to a non occupied row of the screen array as well as when the row is deleted.

Querix4GL also introduces a number of methods which can be used with the DYNAMIC ARRAY data type. A method should be prefixed by a variable of the DYNAMIC ARRAY data type and separated from it by a full stop, i.e.:

```
variable.Method()
```

These methods serve as auxiliary tools used to manage the array effectively. The following methods can be used:

- GetSize() - returns the size of a single-dimensional array.
- Resize() - resizes a single-dimensional array to the given size.
- Delete() - deletes the array element or elements with the specified indexes.
- Append() - adds a new element to the end of the array.
- Insert() - inserts a new array element at the given position.
- Clear() - deletes all the elements from the array.

The methods in details are described in the "Lycia Built-in Functions Guide".

Informix Compatibility

Dynamic arrays support the statements used in Informix 4GL for managing dynamic arrays for compatibility reasons. Querix recommends not to use these statements with the dynamic arrays created with Querix4GL, because this will limit the dynamic resizing of the arrays and may lead to the loss of data. The ALLOCATE ARRAY, DEALLOCATE ARRAY, RESIZE ARRAY statements are supported and function in the same way as they do in Informix 4GL. It is advisable to use the methods provided by Querix instead, which will make dynamic arrays more flexible.



FLOAT

The FLOAT data type is a number data type that is used to store values in the format of double-precision floating-point binary numbers that can have 16 significant digits or fewer. A FLOAT data type requires 8 bytes to store a value.

The actual precision of the data of FLOAT data type depends on the compiler and on the data, but generally it is possible to declare a whole number in the range of 1 to 14 as the precision for a FLOAT data type. The FLOAT data type is generally used to store some scientific data that do not require high precision and can be calculated approximately. To store precise data, use the DECIMAL data type.

The data entered into a table column of FLOAT data type can differ from the data Querix 4GL will display, as the FLOAT data type stores only most significant digits. Such error occurs because the floating-point numbers are stored internally in binary numeration format. If you enter '1.1' into a table column of FLOAT data type, Querix 4GL will display it as 1.09999999. The exact floating-point binary representation of the decimal numeration value may require an infinite number of digits, whereas only a finite number of digits can be physically stored. So an approximate number is stored. Statements of Querix 4GL can specify FLOAT values as floating-point literals. A floating point literal has the following structure: (-)m.mE(-)e: 'm' represents a digit belonging to the mantissa, 'e' represents a digit belonging to the exponent, (-) are the mantissa and exponent signs that are optional. The omitted mantissa or exponent sign will be treated as positive by default. In the floating-point literals, you can use 'E' (either upper or lower case) as the exponent symbol.



If another number format is entered into FLOAT field, or of it is supplied by the program, Querix 4GL makes an attempt to convert it into the FLOAT data type. You can use the USING operator to format the display of the numbers of the FLOAT data type. By default, two decimal places will be displayed, if no formatting has been performed.



FORM

The FORM data type serves as an analogue for a static form identifier and makes the DISPLAY FORM statement more dynamic. It is a special Querix 4GL data type created to make forms the dynamic objects which can be manipulated, it belongs to the Object data types. Like any other variables it can be declared with different scope of reference, which gives a form more flexibility. FORM variables may be passed to and from functions.

A variable of the FORM data type can serve as a dynamic form identifier. It cannot be assigned values directly using the LET statement. Assigning values with the help of the LET statement or in any other way may cause unexpected behaviour. Variables of this data type could be used with the special methods developed for them, which are called in the same way as a function.

The methods used with these variables allow the programmer to open, display, close a form as well as get its parameters. The methods are:

- Open()
- Display()
- Close()
- GetWidth()
- GetHeight()

They resemble traditional 4GL functions, but are actually methods which are used together with a FORM variable and must be separated from it by a full stop. Here is the syntax:

```
variable.Mathod()
```

In this case the name of the variable is at the same time the name of the form. The usage of the methods listed above is described in details in "Lycia Functions Guide". Forms can still be opened closed and displayed using the statements and static form identifiers as well as using the built-in functions.



INT

A synonym for the INTEGER data type.



INTEGER

Whole numbers can be stored using INTEGER data type. The range of values that can be stored using this data type is from -2,147,483,647 to +2,147,483,647. The negative number -2,147,483,648 cannot be used because it is a reserved value. All the values declared as INTEGER are stored as binary integers; they need 4 bytes of memory to be stored. They always have a zero scale. This data type is generally used to store counts, quantities and any other values that can be stored using whole numbers.

This data type can be used in arithmetic operations, because they do not usually cause rounding errors. These operations are performed more efficiently with the data of INTEGER data type if compared to the DECIMAL and FLOAT data types. However, the fractional part of a number assigned to the INTEGER data type is always discarded.

INTEGER data type is used to store SERIAL values used in a database. 4GL automatically assigns the next whole number to the column defined as SERIAL when a row is inserted into a table. The SERIAL number cannot be changed once assigned and it cannot be entered by a user.



INTERVAL

The INTERVAL is a numeric data type that stores periods of time. It stores the difference between two moments in time as the DECIMAL format that includes a contiguous sequence of values representing units of time. The INTERVAL data types can be divided into two types:

- A year-month interval used to represent a span of years, months or both
- A day-time interval used to represent span of days, hours, minutes, seconds and fractions or all of them.

The above two types of intervals cannot be inter-converted within the 4GL, they require different INTERVAL qualifiers and numeric interval values that cannot be mixed.

The INTERVAL data type can be used to store age, sums of ages, periods of some activities, person-hours or person-years of effort, etc. This data type has the structure similar to that of the DATETIME data type. The INTERVAL data type has the following structure when declared:



Element	Description
INTERVAL Qualifier	Specifies the precision and scale of a INTERVAL value.

```
DEFINE my_interval1 INTERVAL YEAR TO MONTH --year-month INTERVAL
DEFINE my_interval2 INTERVAL DAY TO FRACTION --day-time INTERVAL
```

Unlike DATETIME data types, values of the INTERVAL data type can be equal to zero or they can obtain negative values. E.g. if you subtract a larger interval from a smaller one, you can get a negative value of INTERVAL data type as the result.

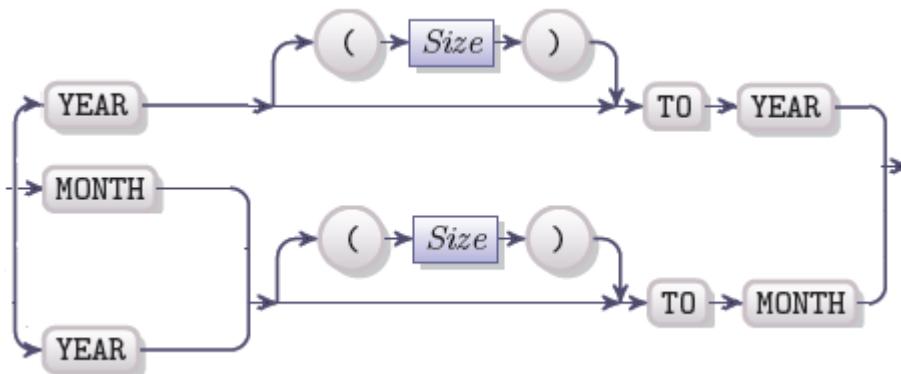
The INTERVAL Qualifier

An INTERVAL qualifier contains information about the largest unit of time used in the INTERVAL value and about the smallest one.

	Note: The <i>first</i> time unit (before the TO keyword) must always be larger than the <i>last</i> one (after the TO keyword).
--	--

The INTERVAL qualifier has the following structure:

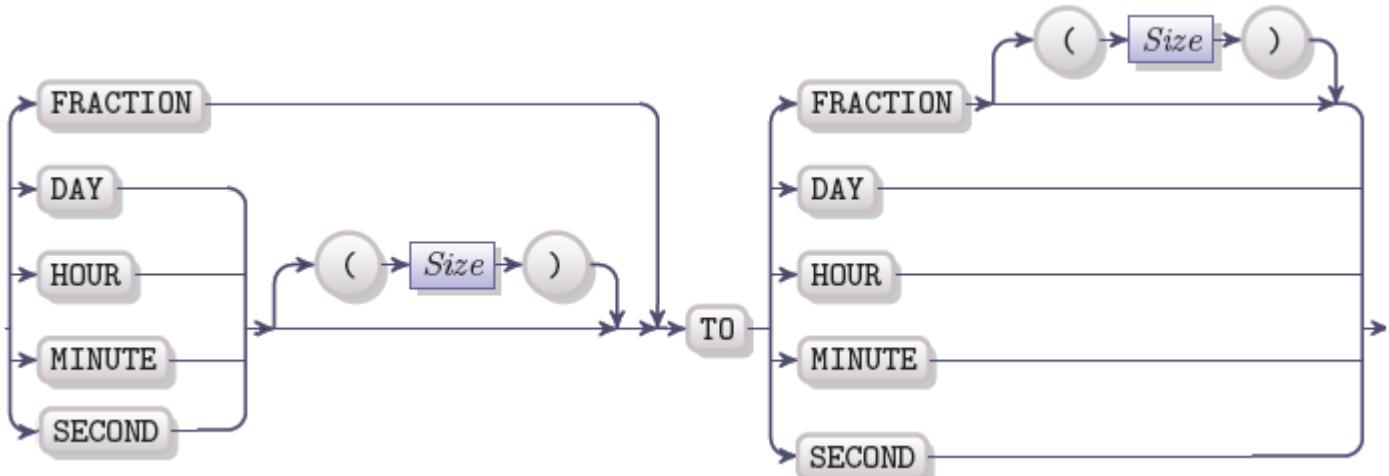
- The INTERVAL qualifier for the *year-month* INTERVAL type has the following structure:



Keyword	Unit of time
YEAR	Specifies that years are used as the largest or the smallest time unit used to measure the elapsed time
MONTH	Specifies that months are used as the largest or the smallest time unit used to measure the elapsed time
Size	An integer from 1 to 9 that specifies the precision of the first INTERVAL qualifier

E.g. `DEFINE inter1 INTERVAL YEAR TO YEAR`

- The INTERVAL qualifier for the *day-time* INTERVAL type has the following structure:



Keyword	Unit of time
DAY	Specifies that days are used as the largest or the smallest time unit used to measure the elapsed time
HOUR	Specifies that hours are used as the largest or the smallest time unit used to measure the elapsed time
MINUTE	Specifies that minutes are used as the largest or the smallest time unit used to measure the elapsed time
SECOND	Specifies that seconds are used as the largest or the smallest time unit used to measure the elapsed time



FRACTION	Specifies that fractions of a second are used as the largest or the smallest time unit used to measure the elapsed time
Size	An integer from 1 to 9 (from 1 to 5 for FRACTION qualifier) that specifies the precision of an INTERVAL qualifier

```
DEFINE inter2 INTERVAL HOUR TO FRACTION(3)
```

	Note: It will cause a compile-time error, if you try to mix qualifiers of year-month and day-time types of interval. The qualifier MONTH TO DAY and other mixing qualifiers are not valid.
--	---

The time unit in the last INTERVAL qualifier may be of the same as the time unit in the first one, but it cannot be a larger time unit. Thus HOUR TO DAY and MONTH TO YEAR are **not** valid qualifiers.

	Note: To specify only one unit of time as a qualifier, you need to specify it as the first and the last keyword: YEAR TO YEAR. It will cause a compile-time error if you try to specify the time unit only once and omit the keyword TO.
--	---

Arithmetic expressions that use year-month INTERVAL values together with DATE values or DATETIME values that do not contain time units smaller than MONTH can return invalid results (like February 30, or June 31). E.g.:

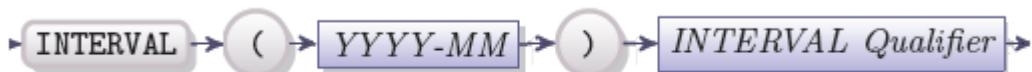
```
DEFINE
    int1 INTERVAL MONTH(4) TO MONTH,
    int2 INTERVAL FRACTION TO FRACTION(4),
    int3 INTERVAL DAY(5) TO FRACTION(2)
```

An INTERVAL value can be specified as an INTERVAL literal or as an INTEGER character string.

The INTERVAL Literal

The INTERVAL literal of INTERVAL data type consists of the INTERVAL keyword, numeric interval value enclosed in parentheses and INTERVAL qualifier:

- The INTERVAL literal for the *year-month* INTERVAL type has the following structure:



Element	Description
YYYY	The numeric indication of years
MM	The numeric indication of months
- (Hyphen)	A delimiter used between years and months
INTERVAL qualifier	The qualifier used for declaration of the year-month interval type



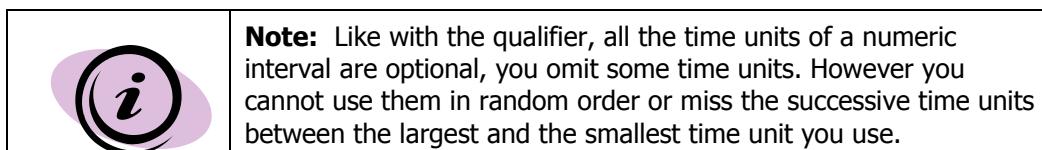
For example, an INTERVAL with the INTERVAL qualifier YEAR(2) TO MONTH can have the following realization:

```
LET inter5 = INTERVAL (87-03) YEAR(2) TO MONTH
```

- The INTERVAL literal for the *day-time* INTERVAL type has the following structure:



Time unit	Description
DD	The numeric indication of days
Space	A space is required between day and hour values
hh	The numeric indication of hours
mm	The numeric indication of minutes
ss	The numeric indication of seconds
: (Colon)	A delimiter used between hours and minutes and between minutes and seconds
fff	A numeric indication of the fractions of a second
. (Dot)	A delimiter which separates fractions from seconds
INTERVAL qualifier	The qualifier used for declaration of the day-time interval type



If you specify the size for the first INTERVAL qualifier, it defines the number of digits which can be stored in the first time unit. An interval declared with the DAY (5) TO MINUTE precision can store up to 999 days.

An INTERVAL qualifier defined as DAY(6) TO HOUR can have the following realization:

```
LET inter3 = INTERVAL(102310 21:46) DAY(6) TO HOUR
```

All the time units except for the FRACTION can have up to 9 digits of precision (FRACTION can have up to 5 digits). When you specify an INTERVAL literal you may explicitly define the number of significant digits you are entering. If you assign a non-default size and do not specify the precision explicitly, a compile-time error will occur. E.g. If you want to enter 103,25 days for the variable declared as INTERVAL DAY TO HOUR, you must enter:

```
LET inter4 = INTERVAL(103 06) DAY(3) TO HOUR
```

It is not necessary to specify every time unit from the declared data type, you can specify only units you need. E.g. if you have declared an INTERVAL variable with DAY TO MINUTE precision, any values can be assigned to it provided that they do not exceed the declared precision. This means you can assign a DAY TO HOUR, DAY TO DAY, HOUR TO MINUTE, HOUR TO HOUR, or MINUTE TO MINUTE values to it, but values that include SECOND and FRACTION will not be valid for a variable declared with DAY TO MINUTE precision.



You need to specify only those time units in the numeric value which correspond to the qualifier you use within the INTERVAL literal. You cannot use them in random order or miss the successive time units between the largest and the smallest time unit you use.

```
INTERVAL(45 20:15:00.234) DAY TO FRACTION -- the correct INTERVAL literal  
INTERVAL(45 00.234) DAY TO FRACTION -- the incorrect INTERVAL literal
```



Note: If you specify the precision of the first time unit, you need to use the number of digits specified. You **cannot** specify DAY(5), but include 4 or less digits in the day value. If you still need to specify a value which is less than the precision, place zeros before the value to match the precision. E.g. : INTERVAL(00045 05:23) DAY(5) TO MINUTE.

The Numeric Interval Value

An INTERVAL value can be also specified as a character string. It must be used without INTERVAL keyword and it enclosed in double quotation marks (" ") or in single quotation marks (' '). The INTERVAL qualifier should also be omitted, though the delimiters must be used. The example below represents a period of time 12 years and 8 months long:

```
LET first_stage = "12-8"
```

In the second example the data is inserted into a table. The 'age' column in the example below has been defined as INTEGER YEAR(2) TO YEAR.

```
INSERT INTO client (fname, lname, sex, age)  
VALUES ("John", "Smith", "male", "54")
```

The character string must contain information about all the time units that have been declared in The INTERVAL qualifier of a variable or of a table column. If a time unit is missing, an error will occur.

Data Entry

The only valid format for entering data of INTERVAL data type into a form field is the unquoted character string. It must be a sequence of values for time units with the corresponding delimiters (the delimiters are described in the "INTERVAL qualifier" section above).

The size of the first INTERVAL qualifier can be up to 9 digits except for the FRACTION. The FRACTION can contain up to 5 digits both as the first and as the last time unit. If an interval is declared as FRACTION TO FRACTION, the size can be specified for the first qualifier only.

A user cannot enter data as an INTERVAL literal into a form field of the INTERVAL data type. The INTERVAL literals are allowed only in 4GL statements and in data type declarations of formonly fields in form specification files.

The following formula can be used to calculate the number of bytes needed to store an INTERVAL value (the resulting number of bytes should be rounded up):



$$\frac{\text{total number of digits of all the time units}}{2} + 1$$

For example, if a DAY(3) TO MINUTE INTERVAL qualifier of an INTERVAL value requires a total of 7 digits (3 for a day, 2 for an hour and 2 for a minute), or $((7/2) + 1) = 4.5$ bytes of storage. When rounded up it will be 5 bytes of storage needed for this INTERVAL value.



MONEY

MONEY is a numeric data type used to store currency amounts. The values of MONEY data type are stored as fixed-point numbers that can have up to 32 significant digits. You can use MONEY(p, s) format to specify the precision and scale. The default number of decimal places for MONEY data type is 2, this means that if you declare MONEY(p) data type, it will be stored as DECIMAL($p, 2$). The precision can be from 1 to 32. The MONEY data type cannot be stored as a floating point number.

If no precision and no scale is specified the default parameters are applied; by default the precision is 16, the scale is 2. The largest absolute value that can be stored with the help of the MONEY data type is $10^{p-s} - 10^{-s}$ (where p is the precision and s is the scale).

Values less than 0.5×10^{-s} (where s is the scale) are stored as zeros. It is not possible to define a precision for a form field of MONEY data type, here the default precision will be used, it is smaller than 32 and is equal to (*length of the filed – 2*).

Values of MONEY data type are displayed with the currency symbol, by default it is a dollar sign (\$), and decimal places are separated with the help of the decimal point. No currency symbol should be entered by the user into form fields of MONEY type or used in 4GL statements in literal MONEY values. The display format can be changed using the DBMONEY or DBFORMAT variable. The settings used in these variables override the default settings of the locale.

To calculate the amount of bytes needed to store a MONEY value, use the formula for DECIMAL data type:

$$\frac{precision + 3}{2}$$

For even scale:

$$\frac{precision + 4}{2}$$

For odd scale:

E.g. MONEY(10,2) will need $((10 + 4) / 2) = 7$ bytes of storage.



NUMERIC

The NUMERIC is a synonym for DECIMAL data type.

Please note, when you come across the word numeric in this manual in lower case letters, bear in mind that it always represents the adjective form of the number noun and is not used to represent the NUMERIC data type.



PREPARED

The PREPARED data type is an object data type which stores a prepared SQL statement. It acts similar to a statement identifier used in the PREPARE statement, but has a number of advantages, if compared to it.

A PREPARED variable can be passed to functions and returned by them. It can be declared as any other variable of simple data type, thus it can be local, module or global. Thus while the prepared statement identifier used in the PREPARE statement has the module scope of reference only, a PREPARED variable can have any scope of reference, depending on the place where it is declared.

You shouldn't assign values to a PREPARED variable using the LET statement. This variable should be used with the special methods, which allow you to specify the prepared statement and then to use it in your program. A method is called in the same way as a function, but it need to be prefixed with the PREPARED variable and separated from it by a full stop, i.e.:

```
variable.Method()
```

Here are the methods used with the PREPARED data type:

- Prepare() - prepares the statement for execution in the RDBMS, it takes the SQL statement to be prepared as an argument
- Execute() - executes the statement previously prepared
- Free() - frees the resources associated with the prepared statement
- IsNative() - verifies whether the statement has been prepared natively
- GetStatement() - returns the text of the SQL statement that was prepared
- GetName() - returns the internal name of the prepared statement

All these methods are described in details in the "Lycia Functions Guide".



REAL

The REAL is a synonym for SMALLFLOAT data type.

When you come across the word real in this manual in lower case letters, please note that it always denotes a number expressible as a limit of rational numbers and is not used to represent REAL data type.

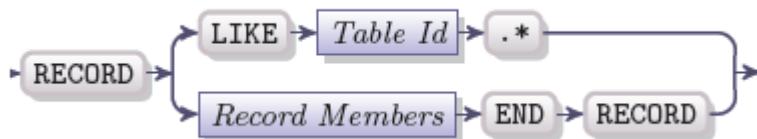


RECORD

The RECORD data type represents an ordered set of members. The set is called a program record, its members are variables of any 4GL data type; they can be used in any combinations to construct a record, the order in which they are included into a record is fixed. Members of a record can be of any 4GL data type:

- Simple Data Types
- Large Data Types (BYTE and TEXT)
- Structured Data Types (ARRAY and RECORD)

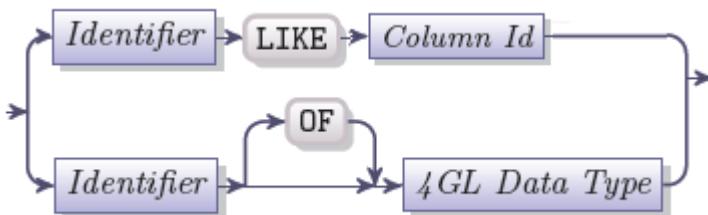
The RECORD data type has the following structure:



Element	Description
Table Id	The name of a database table. It may include other table qualifiers if necessary, such as table owner, server name and database
Record Member	One or more record members separated by commas

Record Member

A member of a record has the same structure as independent declared variables:



Element	Description
Identifier	The name of a record member
Column Id	A name of a database column in the format <i>table.column</i> . It can also be preceded by other table qualifiers: as table owner, server name and database
4 GL Data Type	Any Querix 4GL data type or a user-defined data type

A member can be of any Querix 4GL data type including structured and large data types, it can also be of a user-defined data type. Thus you can have a record member of RECORD data type which will create a nested record:



- A member of a record can be defined of a simple data type. E.g. the example below is a simple record with two members:

```
DEFINE
    my_rec RECORD
        memb1 INT,
        memb2 CHAR(10)
    END RECORD
```

- A record member can be declared LIKE and a table column. This method is used to specify that a *member* of a record has the same data type as a database column.

```
DEFINE
    my_rec2 RECORD
        memb1 LIKE table1.col1,
        memb2 LIKE table2.col2
    END RECORD
```

- As mentioned before, records can include other records as members. Below is an example of a record declared using LIKE keyword and with a member of the RECORD data type (*add_rec* is a member of a record of RECORD data type):

```
DEFINE
    my_rec_sec RECORD
        fname LIKE client.fname,
        lname CHAR(15),
        add_rec RECORD a LIKE orders.list
    END RECORD
END RECORD
```

In case you do not specify the record members and use the LIKE keyword after the RECORD keyword followed by a table name with an asterisk, the record will be created, whose members will be identified like all the columns of the table. A column of SERIAL type will be represented in a record as INTEGER data type.

```
DEFINE my_rec RECORD LIKE order_rec.*,
```



	<p>Note: If <i>table</i> is a view, the column cannot be based on an aggregate. You cannot specify <i>table.*</i> if <i>table</i> is a view that contains an aggregate column</p>
---	--

If the locales of the client and the database do not coincide, do not use the LIKE keyword to declare a member of a record with the name the same as the name of a database column, whose name includes any non-ASCII character not supported by the client locale.

Referencing Record Members

A record can be referenced as the whole or any member of it can be referenced separately. Here are several ways of referencing a record and its members:

- A record can be referenced as a single unit, to do so use the record name with an asterisk: *record.**
- You can use the following notation to reference a record member: *record.member* where *record* is the name of the record that was used in the DECLARE statement and *member* is the name of the record member you want to reference.
- To reference a number of record members in the set order use the following notation: *record.first THRU record.last*, where *record.first* is the name of the first member in the list you want to reference, *record.last* is the name of the last member in the referenced list, THRU is a keyword meaning through, its synonym is THROUGH.

There are some restrictions you need to keep in mind when referencing records using the THRU/THROUGH keyword and the asterisk notation:

6. THRU / THROUGH structure and *record.** cannot be used within a quoted string in PREPARE statements (in SELECT or INSERT clause). It is still possible to use *record.** in an INSERT or SELECT clause of a DECLARE statement.
7. THRU / THROUGH structure cannot be used to indicate a number of screen record members that should be displayed or used for entering data in a form.
8. THRU / THROUGH structure and *record.** cannot be used for referencing a record that contains a variable of ARRAY data type. Though it can be used to reference a record which contains other records as members.

The END RECORD Keywords

The END RECORD keywords must be used to specify the end of the record declaration, if a record contains one or more members. However, if you use the RECORD LIKE keywords to declare a record like all the columns of a specified database table, the END RECORD keywords are not required and will cause a compile-time error.



SMALLFLOAT

The SMALLFLOAT is a numeric data type that is used to store values in the format of single-precision floating-point binary numbers. They can have up to 8 significant digits and usually require 4 bytes for storage.

The SMALLFLOAT, as well as FLOAT data type, is generally used to store data that do not require high precision and can be calculated only approximately. The data entered into a table column of SMALLFLOAT data type can differ from the data 4GL will display, because the SMALLFLOAT data type stores only most significant digits. Such error occurs because the floating-point numbers are stored internally in binary format.

If you enter '1.1' into a table column of SMALLFLOAT data type, Querix 4GL will display it as 1.1000001. The floating-point binary for the value entered using decimal format may require an infinite number of digits, whereas only a finite number of digits can be physically stored. So an approximate value is stored.

A SMALLFLOAT value can be specified as a floating-point literal with the following structure:

mantissa sign exponent sign

-689.03 e-5
mantissa exponent

The mantissa and exponent signs are optional. If they are omitted, they will be treated as positive by default. In the floating-point literals you can use 'E' (either upper or lower case) as the exponent symbol.

If another number format is entered into a SMALLFLOAT field, or of it is supplied by the program, Querix 4GL makes an attempt to convert it into the SMALLFLOAT data type. You can use the USING operator to format the numbers of the SMALLFLOAT data type. Two decimal digits will be displayed by default.



SMALLINT

A SMALLINT data type is a data type similar to INTEGER but with the narrower range of allowed values. It is also stored as a binary integer but requires only 2 bytes for storage due to its small range of values. The value that can be assigned to the SMALLINT data type must be whole numbers that range from -32,767 to +32,767. All the decimal places, if any, are discarded. The negative value -32,768 cannot be used in a database column of a SMALLINT type and cannot be entered to a form field defined as SMALLINT, because this value is reserved.

A value whose actual value is less than 2^{15} can be stored in a variable of SMALLINT data type. This data type is generally used to store, display and manipulate any values that can be represented as whole numbers, these are:

- Small whole numbers
- Values of Boolean expressions
- Measurements, etc.

Due to the fact that only 2 bytes are required to store a SMALLINT value, highly efficient arithmetic operations can be performed with values of this type.



STRING

The STRING data type provides a variant holder for the character strings. It does not require the size to be declared; the size of a STRING variable changes depending of the size of the value stored in it.



Note: The variables of the STRING data types cannot be used in the SQL expressions for receiving the values of a query, since the size of the receiving variables needs to be set before the values are returned.

The STRING values can be compared with a CHAR, VARCHAR or STRING value. No blank spaces are added to or stripped of a variable of the STRING data type. Unlike VARCHAR variables, STRING variables can contain an empty string without being NULL. Thus, if you used the trim() built-in function or the CLIPPED operator against a STRING variable containing only whitespaces, the variable will contain an empty string which will not be NULL. The IS NULL operator used with such a string will return FALSE. A VARCHAR variable in this case will be evaluated to NULL. However, STRING variables are still initialized to NULL when they are declared.

Like VARCHAR variables, STRING variables store significant trailing whitespaces added to the value explicitly and does not add whitespaces at the end of values automatically like the CHAR variables do. Thus strings "abcd " and "abcd" stored in STRING variables will be different when compared.

The size of such variable can be increased or decreased dynamically during runtime, depending on the length of the value assigned to it. When no value is assigned to a STRING variable, it has the length of a single character.



Note: Usually the STRING data type requires 1 byte per character, so the size in bytes is equal to the size in characters, but in some East Asian locales, a character may require more than one byte to be stored. Some white space characters can also occupy more than one byte.



TEXT

Character strings can be stored using TEXT data type. The TEXT data type is similar to the BYTE data type, but whereas BYTE data type can store any information that can be stored in digital form, the TEXT data type can store strings of ASCII printable characters. It can also include the following whitespace characters:

- TAB (CONTROL-I)
- NEWLINE (CONTROL-J)
- FORMFEED (CONTROL-L)

Some locales support other whitespace characters. However these characters are not included to the characters supported by default, so if any other non-printable character that is not listed above is included into the TEXT value, unexpected result may occur and the TEXT value may be processed improperly.

TEXT values have a theoretical size limit of 2^{31} bytes, although the practical limit is determined by the available system storage. In East Asian locales, more than one byte of storage space may be required per character.

A TEXT variable is usually used to manipulate a database column of TEXT type or to display a text file using a text editor. After a variable of TEXT data type has been declared, the LOCATE statement should be used to inform the program about the location of the information stored in this variable.

A value retrieved from a column of TEXT type is completely or partially assigned to a variable of TEXT data type. To assign only a part of the whole TEXT value, use square brackets, e.g.:

```
SELECT treaty_text [1,245] INTO treaty FROM doc_tab
```

The above example results in retrieving the first 245 bytes from the text of the treaty and saving them to the variable *treaty*. In most locales one byte is used to store one character, there are some whitespace symbols that require more space but in general this will result in retrieving the first 245 symbols from the text.

Restrictions applied to TEXT Variables

There are a number of restrictions which are applied to a value of the TEXT data type:

- A formonly field of the TEXT data type or a field that represents a database column of TEXT data type can display only small amount of text, the limit is set by the length of such field or by the declared size of a column. To view and edit the complete text, you must assign PROGRAM attribute to such field.
- The WORDWRAP attribute can be used to display the whole text or part of it depending on the total length of all the segments, although the text cannot be edited in such a way.
- Only three attributes can be used with a form field of the TEXT type, these are: PROGRAM, WORDWRAP and COLOR.
- TEXT values are passed by reference in CALL, RETURN and OUTPUT TO REPORT statements. The DISPLAY TO and DISPLAY (console)statement can be used to display the TEXT value, whereas DISPLAY AT statement can also be used, but will display only one line of text, the rest will be truncated.
- You can assign the following values to a TEXT variable: NULL or values of CHAR, TEXT, VARCHAR, STRING data types.



TINYINT

The TINYINT is an integer data type used for storing small values. The range of values that can be stored using this data type is from -128 to 127. All the values declared as TINYINT are stored as binary integers; they need 1 byte of memory to be stored. They always have a zero scale.

This data type can be used in arithmetic operations, because they do not usually cause rounding errors. This data type is used for performance efficiency, if you need to store small numbers.

The TINYINT variables are initialized to 0 when they are declared. They can never be initialized to NULL.



VARCHAR

The VARCHAR data type is similar to the CHAR data type and is also used to store character strings. The size of the VARCHAR variable is limited by 32767 characters. You can declare the maximum size of a VARCHAR variable by using *(size)* parameter. A string stored in this variable can be shorter than the declared *size*, but it cannot be longer. If you do not specify the size explicitly, the default size will be used which is equal to 1.

The *size* parameter of the VARCHAR data type indicates the storage that will be reserved in the system for the variable declared as a VARCHAR. The reserved VARCHAR declaration does not affect the behaviour of the 4GL syntax, but it may affect the behaviour of a database which contains a column declared as VARCHAR.

The VARCHAR values are compared in the same way as CHAR values, spaces are added to the end of the shorter value until both values are equal in lengths. A VARCHAR value can be compared only with a CHAR, VARCHAR or STRING value. If a VARCHAR variable consists only of white spaces, the trim() function or the CLIPPED operator applied to it will change its value to NULL.

There are some other features that differentiate VARCHAR data type from the CHAR data type:

- No trailing blank spaces are ever stripped of a VARCHAR variable or of a value entered into a database column of the VARCHAR data type, if they are entered by a user
- No blank spaces are added to the end of the VARCHAR values to fit the declared size, if they are not added by the user explicitly, so there is no need to use the CLIPPED operator when working with this data type

	Note: Usually VARCHAR data type requires 1 byte per character, so the size in bytes is equal to the size in characters, but in some East Asian locales a character may require more than one byte to be stored. Some white space characters can also occupy more than one byte.
---	--



WEBSERVICE

The WEBSERVICE is an object data type which stores information about a webservice. Due to this data type the developer does not need to use web service handlers. It reduces the number of code required to operate a web service. This data type is also used, if you create a webservice client automatically by means of Querix tools.

A WEBSERVICE variable can be declared like any other simple data type. Depending on the declaration context it can have any scope of reference. It can be passed to and from functions like variables of other data types. You should not use the LET statement to assign data to it. You should use the special methods instead. The methods are called in the same way as other functions and are used together with the CURSOR variable in the following manner:

```
variable.Method()
```

Here are the methods typically used with the WEBSERVICE data type:

- LoadMeta() - loads the specified WSDL meta data definition file
- SelectOperation() - specifies the web service operation you want to invoke
- Execute() - executes the selected operation

All these methods are described in details in the "Lycia Functions Guide". They are also used in an automatically generated file of a webservice client.



WINDOW

The WINDOW object data type describes a 4GL window. If a variable of this data type used to open and manipulate a window, it becomes the subject of the scoping rules for 4GL variables. A window object declared implicitly by using the OPEN WINDOW statement has the global scope, whereas a WINDOW variable can have any scope of reference depending on the place where it is declared. Thus a window opened using a WINDOW variable can have any scope of reference.

A WINDOW variable is declared in the same was as variables of simple data types. It can be passed to functions as an argument and returned by them. You should not assign values to these variables using the LET statement. You should use the special methods instead to open and manipulate windows. A method should be prefixed by a variable of the WINDOW data type and separated from it by a full stop, i.e.:

```
variable.Method()
```

Here are the methods typically used with the variables of the WINDOW data type:

- Open() - opens a window with the desired parameters which are used as the arguments
- OpenWithForm() - opens a window with the form specified as its argument
- Close() - closes a window opened by the Open() method
- Move() - changes the position of an open window
- Resize() - changes the size of an open window
- DisplayAt() - displays text to an open window
- Message() - displays a message at the message line of the window
- Error() - displays an error at the error line of the window

These methods are described in details in the "Lycia Built-in Functions Guide".



Conversion of Data Types

4GL performs data type conversion automatically, if it makes sense and does not contradict the rules of transformation. Such values as numbers, character strings and values that indicate time can be assigned to a variable of a different but compatible data type. If a number value is assigned to a variable of CHAR type, it will be converted into a character string, because these data types are fully compatible when converting from numeric to character data type. However the reverse conversion does not always make sense. If you try to use a character string which consists only of digits, e.g. "999.01" and which is of CHAR data type in an arithmetic expression, Querix 4GL will be able to convert such string into a number. The conversion will fail if you try to use a character string that contains letters and other symbols that are not digits or mathematic signs in similar situation.

Number to Number Conversion

Conversion from one numeric data type to another must be paid attention to while the receiving data type must be capable to store all the value which is being converted. In the process of conversion some transformations are performed to adjust the value so that it could be stored as the target data type. E.g. if you try to convert a DECIMAL data type to the INTEGER data type, decimal places will be discarded and the information stored in them will be lost.

There are also situations that will not only cause the loss of information but may also cause the conversion to fail. E.g. if you try to convert FLOAT to SMALLFLOAT or INTEGER to SMALLINT the conversion will fail, if the value you are trying to convert has more than 8 significant digits or is larger than 32,767 (respectively).

The similar overflow may occur if you try to convert FLOAT or SMALLFLOAT values to INTEGER, SMALLINT, or DECIMAL data types. E.g. converting a FLOAT value to a DECIMAL value may result in overflow, underflow or rounding errors, because a floating-point number needs to be rounded off before it could be stored as a fixed point number.

Arithmetic Operations and Number Conversion

The most of the arithmetic operations in Querix 4GL are performed using DECIMAL values, because most of the data types are stored internally as DECIMAL values.

The result of the arithmetic operations on two numbers is always a DECIMAL but the scale and precision of the DECIMAL depends on the source data types:

Source Data Type	An Arithmetic Expression	Resulting Data type
FLOAT	→	DECIMAL (16)
INTEGER	→	DECIMAL (10, 0)
MONEY (p)	→	DECIMAL (p, 2)
SMALLFLOAT	→	DECIMAL (8)
SMALLINT	→	DECIMAL (5, 0)

	Note: An error occurs, if to produce a result of an arithmetic expression some significant digits are lost due to the limits of the DECIMAL data type or due to other reasons.
--	---



In the expressions that perform addition or subtraction trailing zeroes are added to the scale of that operand, which scale is shorter, until the scales of the both operands have equal number of decimal places. Trailing zeroes, as well as leading ones, do not influence the precision or scale of the resulting DECIMAL value.

If at least one operand is a floating-point number, the result of an expression will be a floating-point number of DECIMAL data type. The precision and the scale of the resulting DECIMAL value depend on the precision and scale of the operands as well as on the arithmetic operator used.

DATE and DATETIME Conversion

A DATE value can be converted to a DATETIME value, as well as a DATETIME value can be converted to a DATE value. If a DATETIME value contains time units less than a day and it is converted to or from the DATE data type, no error will occur, though all the units less than the day will be either ignored or filled with zeros. The exact behaviour depends on the context in which the conversion takes place. Below are several examples of such conversions (the display format of the values is default)

DATE to DATETIME Conversion

- When DATE is converted to DATETIME with YEAR TO DAY Day Time Qualifier, the conversion is performed without information loss. E.g. 04/18/2010 will be converted to 2010-04-18
- When DATE is converted to DATETIME with YEAR TO HOUR / MINUTE / SECOND / FRACTION Day Time Qualifier, all the time units smaller than DAY are filled with zeros. E.g. 04/18/2010 will be converted to 2010-04-18 00:00:00 when the target DATETIME data type has the YEAR TO SECOND qualifier.

DATETIME to DATE Conversion

- When DATETIME with YEAR TO DAY qualifier is converted to DATE, the conversion is performed without information loss. E.g. 2009-01-31 will be converted to 01/31/2009
- When DATETIME with YEAR TO HOUR / MINUTE / SECOND / FRACTION qualifier is converted to DATE, all the time units smaller than DAY will be discarded. E.g. 2009-01-31 10:39:23 will be converted to 01/31/2009 when the source DATETIME data type has the YEAR TO SECOND qualifier.
- When a DATETIME value misses larger time units like a year or a month and if it is converted to the DATE data type, the missing time units are retrieved from the system clock-calendar, the time units smaller than DAY will be discarded. E.g. DATETIME value with MONTH TO DAY Day Time Qualifier 12-24 will be converted to 12/24/ <the current year>.
- When a DATETIME value contains no time units that are common for the DATETIME and DATE data types, conversion is still possible, though the resulting DATE data type will be the current date according to the clock-calendar. All the time units smaller than DAY will be discarded E.g. DATETIME value with HOUR TO SECOND Day Time Qualifier 23:15:37 will be converted to <present day>/<current month>/<current year>.

Conversion of CHARACTER Value to DATETIME or INTERVAL Value

The DATETIME and INTERVAL values may be entered as literals and as character strings. Values that are specified as character strings and with strict adherence to the format of time data types are automatically converted to DATETIME or INTERVAL values, e.g.:

- The below examples will have the same result and will indicate a moment of time defined as August 30, 2009:

```
LET curr_time = DATETIME(09-08-30 23:15) YEAR TO MINUTE --DATETIME literal
```



```
LET curr_time = "09-08-30 23:15" --character string
```

- The below examples will have the same result and will indicate a period of time a little more than 10 days long:

```
LET interv = INTERVAL(10 03:19:55) DAY TO SECOND --INTERVAL literal  
LET interv = "10 03:19:55" -- character string
```

The information about all the time units declared must be included to a character string which represents a DATETIME or INTERVAL value. If a character string does not contain information about all the declared time units, an error will be returned. If you do not want to enter all the time units declared, you must use DATETIME and INTERVAL literals, because 4GL supplies missing time units for the literals but it does not supply the missing values for character strings. Character strings with missing values will produce errors. E.g. if you want to enter only hours and minutes for an INTERVAL declared with DAY TO SECOND Day Time Qualifier, use INTERVAL literal:

```
LET my_int = INTERVAL(12:45) HOUR TO MINUTE
```

The below example which uses a character string in such situation is **invalid**:

```
LET my_int = "12:45"
```

The above example is invalid because the character string does not specify the days and the seconds though they are present in the data type declaration.

Empty character strings are converted to null values.

Conversion between Character Strings and Numbers

Character data types (CHAR, VARCHAR and STRING) can be converted to numeric data types and vice versa if the character strings do not contain symbols that are not allowed for numeric data types (e.g. letters).

If a numeric data type is converted to a character string (e.g. using LET statement), the currency symbols and delimiters will be locale-specific rather than default.

Large Date Types Conversion

A value of the TEXT data type can be converted to BYTE data type. It is the only type of conversion available for large data types in Querix 4GL.



Summary on Simple Data Type Conversion

As discussed above, some data types in Querix 4GL are compatible. This means they are automatically converted by 4GL. Nevertheless the error-free conversion is dependent on the data types that are being converted.

This table specifies the compatible data types (the symbols used in the table are explained below it):

Resulting Data Types	Source Data Type												
	CHAR	VARCHAR	INTEGER	SMALLINT	FLOAT	SMALLFLOAT	DECIMAL	MONEY	DATE	DATETIME	INTERVAL	BOOLEAN	STRING
CHAR	1	1	1	1	1	1	1	1, 8	1, 9	1	1		1
VARCHAR	1	1	1	1	1	1	1	1, 8	1, 9	1	1		1
INTEGER	2, 3	2, 3			3, 4	3, 4	3, 4	3, 4	11	NO	NO		2, 3
SMALLINT	2, 3	2, 3	3		3, 4	3, 4	3, 4	3, 4	3, 11	NO	NO		2, 3
FLOAT	2, 3, 5	2, 3, 5	10	10			3	3	11	NO	NO		2, 3
SMALLFLOAT	2, 3, 5	2, 3, 5	10, 5	10	5		3, 5	3, 5	11, 5	NO	NO		2, 3, 5
DECIMAL	2, 3, 6	2, 3, 6	3	3	3, 6	3, 6	3, 6	3, 6	3, 11	NO	NO		2, 3, 6
MONEY	2, 3, 6	2, 3, 6	3	3	3, 6	3, 6	3, 6	3, 6	3, 11	NO	NO		2, 3, 6
DATE	2	2	11	11	3, 4, 11	3, 4, 11	3, 4, 11	3, 4, 11		12, 14	NO	NO	2
DATETIME	2	2	NO	NO	NO	NO	NO	NO	13, 14	7, 14	NO	NO	2
INTERVAL	2	2	NO	NO	NO	NO	NO	NO	NO	NO	3, 7	NO	2
BOOLEAN													
STRING								8	9				

A cell represents the result achieved when a data type from the row above is converted to a corresponding data type from the column on the left. The cells marked with 'NO' word indicate that 4GL does not support automatic conversion for the given pair of the data types.

The cells shaded with blue colour indicate the data conversion which is done without restrictions and violations.

	Note: For detailed information about the conversion rules for the BOOLEAN data type see the " BOOLEAN " section of this reference
---	--

The numbers in the cells represent the possible errors and failures that can occur while converting the corresponding data types.



Causes of Conversion Failure

These are the numbers of errors that indicate possible conversion failures:

Error number	Description
1	Occurs when the resulting data type value is longer than the size of the receiving data type variable. E.g. if a resulting character string of CHAR type is longer than CHAR(size) declared. The characters to the right that do not fit the receiving variable will be discarded
2	A character string value of the source data type must be in the form of a literal of the resulting data type
3	An overflow error may occur if the resulting value exceeds the permitted range of the receiving data type
4	A part of the resulting value may be truncated, e.g. decimal places are truncated when converting from DECIMAL to INTEGER
5	Less significant digits may be discarded, if the resulting data type is restricted to fewer significant digits than the source data type
6	Lower-order fractional digits are discarded if the resulting data type is restricted to fewer decimal places
7	Some parts of the value can be discarded due to differences in qualifiers

Causes of Conversion Violation

These are the numbers of errors that indicate some features that do not normally cause conversion failure but that may produce unexpected results:

Error number	Description
8	The format of the resulting value is determined by DBMONEY or DBFORMAT environment variable
9	The format of the resulting value is determined by DBFORMAT, DBDATE, or GL_DATE environment variable
10	A rounding error can occur during conversion
11	The days count is assigned as an integer value
12	During DATETIME to DATE conversion an implicit EXTEND statement is executed which extends the DATETIME value to YEAR TO DAY format
13	During DATE to DATETIME conversion the DATE is transformed to DATETIME YEAR TO DAY literal
14	Any missing time units within the resulting data type will be retrieved from the system clock-calendar



Data Types Used with Databases

Not all of the above listed data types can be successfully used to create tables in databases. This manual does not cover the SQL statements embedded into 4GL which allow the programmer to create and manipulate the database structure. However, this section of the manual will specify which data types can be used for database manipulations and which cannot.

Most of the data types (except the object data types) can be used to store the values returned by a database query. Still there are some restrictions which are specified in the table below. The table below presumes that the values retrieved by queries are compatible with the data types of the used variables. The data types compatibility is described above.

Data Type	In Creating Tables	In Queries	In 4GL
ARRAY	No. Columns cannot be of structured data types.	Yes, array elements can be used as the receiving variables, but not the array as a whole.	Yes.
BOOLEAN	No.	Yes, if the retrieved value is of a numeric data type or of the DATE data type and does not exceed the limit for the SMALLINT data type.	Yes.
BIGINT	Yes, if your database supports this data type. (E.g. Informix 7.x does not support it, whereas Informix 11.x does)	Yes, if your database supports this data type. (E.g. Informix 7.x does not support it, whereas 11.x does). The data type of the value retrieved should be compatible with integer data types	Yes.
BYTE	Yes.	Yes.	Yes.
CHAR	Yes.	Yes.	Yes.
CURSOR	No.	No.	Yes.
DATE	Yes.	Yes.	Yes.
DATETIME	Yes.	Yes.	Yes.
DECIMAL	Yes.	Yes.	Yes.
FLOAT	Yes.	Yes.	Yes.
DYNAMIC ARRAY	No. Columns cannot be of structured data types.	Yes, array elements can be used as the receiving variables, but not the array as a whole.	Yes.
FORM	No.	No.	Yes.
INTEGER	Yes.	Yes.	Yes.
INTEGER8	Yes, if your database supports this data type. (E.g. Informix 7.x does not support it, whereas Informix 11.x does)	No.	No.
INTERVAL	Yes.	Yes.	Yes.



MONEY	Yes.	Yes.	Yes.
NCHAR	Yes.	Yes.	Yes.
NVARCHAR	Yes.	Yes.	Yes.
PREPARED	No.	No.	Yes.
RECORD	No. Columns cannot be of structured data types.	Yes, record members and whole records can receive retrieved values, if the data types and number of values are compatible.	Yes.
SERIAL	Yes.	No.	No.
SERIAL8	Yes, if your database supports this data type. (E.g. Informix 7.x does not support it, whereas Informix 11.x does)	No.	No.
SMALLFLOAT	Yes.	Yes.	Yes.
SMALLINT	Yes.	Yes.	Yes.
STRING	No.	No.	Yes.
TEXT	Yes.	Yes.	Yes.
TINYINT	No.	No.	Yes.
VARCHAR	Yes. But the maximum size of the column is limited by your database settings and is smaller than the maximum size in the 4GL language.	Yes.	Yes.
WEBSERVICE	No.	No.	Yes.
WINDOW	No.	No.	Yes.



CHAPTER 2: 4GL EXPRESSIONS

This chapter will cover the 4GL expressions of all types together with the operators and operands. It will also touch upon the positions the expressions can be used it. It will discuss the data types of an expression result depending on the operands' data types and the operator.



Notion of 4GL Expressions

The notion of a 4GL expression is closely related to the notion of a 4GL data type. A data type characterizes a general category of values. Expressions are used to define specific data values. There are many types of expressions in Querix 4GL. They can be used as arguments in:

- Other expressions
- Statements
- Functions
- Operands
- Form specifications

The syntax of an expression determines which data type it returns. The resulting data type may also depend on the context in which it is used. The 4GL expressions can be divided into 3 major categories depending on the data type of the returned value.

1. **Numeric Expressions** – return a value of any numeric data type (includes Integer expressions and Boolean expressions)
 - a. **Integer Expressions** – return a value of INT or SMALLINT data type
 - b. **Boolean Expressions** – return TRUE or FALSE or NULL (a separate case of an integer expression)
2. **Character Expressions** – return a character string of CHAR, STRING or VARCHAR data type
3. **Time Expressions** – return a value that belongs to INTERVAL, DATE or DATETIME data type

The term 4GL expression is used in this guide to refer any of the above listed types of expressions or all of them together.

Querix 4GL Expressions versus SQL Expressions

The main difference between SQL expressions and 4GL expressions is that SQL expressions (those expressions that are used in the SQL statements) are evaluated by the database server and not by the 4GL. The 4GL expressions use operators that resemble operators used in the SQL expressions and are not identical.

SQL operators can be used only in SQL statements, 4GL operators can only be used in 4GL statements and they cannot be interchanged. The identifiers that belong to SQL (table names, column names, database names, etc.) can be used with the LIKE keyword or in the name of a form field only if they stick to the naming rules used in Querix 4GL.

The **4GL** expressions **do not** accept the following SQL operators and operands:

- SQL identifiers (e.g. column names),
- SQL keywords (e.g. USER, ROWID)
- Built-in SQL functions that are not integrated into 4GL
- SPL variables
- SQL operators (e.g. BETWEEN, IN can be used only in form specifications)
- SQL keywords (e.g. EXISTS, ALL, ANY, SOME)

The **SQL** expressions **do not** accept the following 4GL operators:

- Exponentiation and modulus operators



- String operators (ASCII, COLUMN, SPACE, SPACES, WORDWRAP)
- Field operators (field_touched(), get_fldbuf(), infield())
- Report operators (LINENO, PAGENO)
- Time operators (date(), TIME)

Components of Querix 4GL Expressions

A 4GL expression can include 4GL operators, parentheses, which are used to change the default order of operators, and operands (named values, form field names, literal values, any other 4GL expressions, and calls for functions that return a single value).

Parentheses Used in 4GL Expressions

The parentheses are used to change the default order of the 4GL operators in which they are used. The parentheses are regarded by 4GL as delimiters rather than as operators. Parentheses are also used in 4GL to enclose function arguments and for other purposes.

Here is the example of parentheses used to change the default order of the arithmetic operations and in this way to change the value returned by the expression:

```
LET a = 5+2*3
```

The variable 'a' will be evaluated to '11'.

```
LET a = (5+2)*3
```

The variable 'a' will be evaluated to '21'.

Operators Used in 4GL Expressions

Any of the operators listed below can be used in the 4GL expressions. The order in which the operators will be executed by 4GL depends on the precedence of the operators conditioned by Querix 4GL. The highest precedence (16) means that the operator is the first to be used no matter which operators it is combined with. The operator with the precedence of 15 will be executed before any other operator, except for the one with precedence of 16, if it is present in the expression. The operator with precedence of 1 will be executed after all the other operators present in the same expression. If two operators of the same precedence are used in one expression, order of execution depends on their associativity, which can be right or left, though some 4GL operators are non-associative and are indicated in the table below with the word 'None' in the Associativity column.

The following notations are used in the table below:

- *exp* – any type of expression
- *int-exp* – Integer Expression
- *char-exp* – Character Expression
- *num-exp* – Numeric Expression

Operator	Precedence	Associativity	Description	Example
.	16	Left	Belonging to a program record	rec1.member5
[]			Index or a character substring of an	arr1[a,4,100] [1, <i>int-exp</i>]



			ARRAY	
()		None	Function calling	funct1(arg1, exp)
UNITS	15	Left	An interval of time with one Day Time Qualifier	<i>int-exp</i> UNITS DAY
+	14	Right	Unary plus operator	+ <i>num-exp</i>
-			Unary minus operator	- <i>num-exp</i>
**	13	Left	Exponentiation operator	<i>num-exp</i> ** <i>int-exp</i>
MOD			Modulus operator	<i>int-exp</i> MOD <i>int-exp</i>
*	12	Left	Multiplication operator	a * <i>num-exp</i>
/			Division operator	<i>num-exp</i> / arr2[5]
+	11	Left	Addition operator	<i>num-exp</i> + 6.78
-			Subtraction operator	(a+25) - <i>num-exp</i>
	10	Left	Concatenation operator	"con" <i>char-exp</i>
LIKE	9	Right	Comparison of strings	<i>char-exp</i> LIKE "*td%%"
MATCHES			Comparison of strings	<i>char-exp</i> MATCHES "*ah"
<	8	Left	Less than	<i>exp1</i> < <i>exp2</i>
<=			Less than or equal to	a <= arr[x,7]
= <i>or</i> ==			Equal to	a = <i>exp</i>
>=			Greater than or equal to	a >= rec2.member3
>			Greater than	tel_num_var > <i>exp</i>
!= <i>or</i> <>			Not equal to	rec3.member1 != LENGTH(a)
IN()	7	Right	Testing for the belonging to a set	<i>exp</i> IN (a, 5, <i>exp2</i>)
BETWEEN... AND	6	Left	Testing for the range in WHERE clause	BETWEEN <i>int-exp</i> AND 5
IS NULL	5	Left	Testing that a value is NULL/NOT NULL	a IS (NOT) NULL
NOT	4	Left	Logical inverse	NOT (<i>exp</i> IN (date1, date2))
AND	3	Left	Logical conjunction	<i>exp</i> AND a
OR	2	Left	Logical disjunction	<i>exp</i> OR <i>exp2</i>
ASCII	1	Right	Returns character of ASCII type	LET a = ASCII <i>int-exp</i>
CLIPPED			Deletes trailing whitespaces	DISPLAY var1 CLIPPED
COLUMN			Begins display in the line mode	PRINT COLUMN 35, "15"
ORD			Logical inverse of ASCII operator	LET b = ORD <i>char-exp</i>
SPACES/SPACE			Inserts whitespaces	DISPLAY <i>exp</i> , 1 SPACE, <i>exp</i>
USING			Set non-default format for character string	CURRENT USING "dd/mm/yy"
WORDWRAP			Display text in several lines	PRINT <i>char-exp</i> WORDWRAP

The data type of a value returned by a 4GL expression depends on the type of the expression (on the operators used) and on the data types of the operands used in the expression. The following table contains the information about the resulting value data type depending on the operators (which are listed in the table above) and on the data type of the operands. Querix 4GL operators do not support operands of the structured data types, however, they can be used with individual elements of an array or with members of a record. Most of the operators cannot be used with the operands of the Large data types, except for those for which it is explicitly specified in the table.



4GL Expression	Precedence	Data Type of the left operand (a)	Data type of the right operand (b)	Data type of the returned value
a.b	16	RECORD	Any Simple data type	Corresponds to b data type
array1[a,b]		INT, SMALLINT	INT, SMALLINT	Any Simple data type
(b)		-	Any Simple or Large data type	Any Simple data type
a UNITS DAY	15	INT, SMALLINT	-	INTERVAL
+b	14	-	Numeric data type	Corresponds to b data type
-b				
a**b	13	Numeric data type	INT, SMALLINT	Numeric data type
a MOD b		INT, SMALLINT	INT, SMALLINT	INT, SMALLINT
a*b	12	Numeric data type or INTERVAL	Numeric data type	Numeric data type or INTERVAL
a/b				
a+b	11	Numeric or Time data type	Numeric or Time data type	Numeric or Time data type
a-b				
a b	10	Any Simple data type	Any Simple data type	Character data type
a LIKE b	9	Character data type	Character data type	Boolean (TRUE, FALSE)
a MATCHES b				
a<b	8	Any Simple data type	Corresponds to a data type	Boolean (TRUE, FALSE)
a<=b				
a=b or a==b				
a>=b				
a>b				
a!=b or a<>b	7	Any Simple data type	Corresponds to a data type	Boolean (TRUE, FALSE)
a IN(y)				
BETWEEN a AND b	6	Any Simple data type	Corresponds to a data type	Boolean (TRUE, FALSE)
a IS NULL	5	Any Simple or Large data type	-	Boolean (TRUE, FALSE)
NOT b	4	-	Boolean (TRUE, FALSE)	Boolean (TRUE, FALSE)
a AND b	3	Boolean (TRUE, FALSE)	Boolean (TRUE, FALSE)	Boolean (TRUE, FALSE)
a OR b	2	Boolean (TRUE, FALSE)	Boolean (TRUE, FALSE)	Boolean (TRUE, FALSE)
a CLIPPED	1	Character data type	-	Character data type
ASCII b		-	INT, SMALLINT	
COLUMN b		-	Character data type	
ORD (b)		INT, SMALLINT	-	Character data type
a SPACES		Character data type, MONEY, DATE	Character data type	
a SPACE				
a USING "b"				
a WORDWRAP		Character data type, TEXT	-	



If you use an operator with an operand which data type is not listed in the table for this operator, Querix 4GL will attempt to perform a data type conversion. An error will occur, if the attempt fails.

Bitwise Operators

Querix 4GL also supports bitwise operators. These are used to compare and operate on values based on certain rules. The bitwise operators operate on one or two bit patterns or binary numerals at the level of their individual bits. In 4GL you can use only decimal numbers with these operators. If you try to use binary numbers, they will be treated as decimal numbers and the initial zeros will be stripped. The decimal numbers used with bitwise operators are operated internally as binary numbers, however, an expression which includes a bitwise operator will return a value converted to decimal numeration system.

AND Operator

The AND(&) bitwise operator takes two binary representations of decimal numbers of equal length and performs the logical AND operation on each pair of corresponding bits. If the two bits are 1 then the resulting bit will be 1, otherwise the resulting bit will be 0. The two operands of the AND(&) operator should be of the same length in binary representation.

For example, if we use the AND(&) operator to compare numbers 9 and 12, we will get number 8 as the result:

$$9 \& 12 = 8$$

Internally, the following operation is performed:

1001	-- binary representation of number 9
&	-- AND operator
1100	-- binary representation of number 12
<hr/>	
1000	-- binary representation of number 8

As only the first bits of the both numbers are 1, the first bit of the resulting value is also 1, the rest are 0.

OR Operator

The OR(|) operator takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits. If at least one bit in the pair of bits of the compared operands is 1, the bit of the resulting value will also be 1. Thus the result will be 0 only if both bits from the pair are 0, such pairs of bits as 1 or 1, 1 or 0, and 0 or 1 will result in 1.

For example, if we use the OR(|) operator to compare numbers 9 and 12 we will get number 13 as the result:

$$9 | 12 = 13$$

Internally the following operation is performed:

1001	-- binary representation of number 9
	-- OR operator
1100	-- binary representation of number 12
<hr/>	
1101	-- binary representation of number 13



As only the third pair of bits contains both 0, the rest of the pairs result in 1.

XOR Operator

The XOR(^) operator takes two bit patterns of equal length, and produces another one of the same length by matching up corresponding bits. The resulting value for each pair of bits is 1, if the bits are different, and 0, if they are the same.

For example, if we use the XOR(^) operator to compare numbers 9 and 12 we will get number 5 as the result:

$9^12=5$

Internally the following operation is performed:

1001	-- binary representation of number 9
^	-- XOR operator
<u>1100</u>	-- binary representation of number 12
0101	-- binary representation of number 5

The first and third pairs of bits are the same (1 and 1, 0 and 0), thus they both return 0. The second and fourth pairs of bits are different (0 and 1, 1 and 0) and they return 1.

NOT Operator

The NOT(~) operator is an unary operation that performs logical negation on each bit. It transforms 1 into 0 and 0 into 1.

For example, if we use the NOT(~) operator with number 9, we will get the number 6, which binary representation is opposite to binary representation of number 9:

$\sim 9=6$

Internally the following operation is performed:

1001	-- binary representation of number 9
~	-- OR operator
0110	-- binary representation of number 6

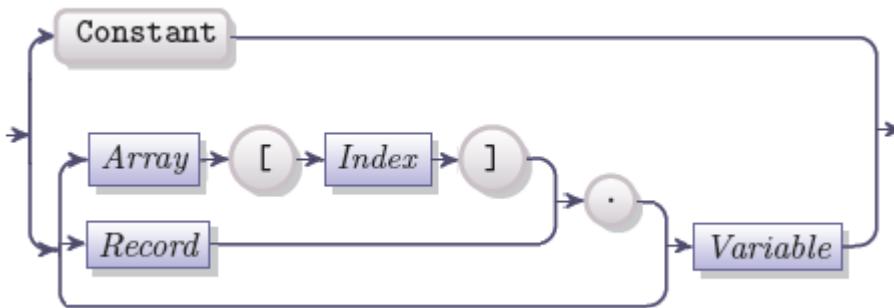
Operands Used in 4GL Expressions

Querix 4GL can use any of the following as operands in 4GL expressions:

- Named values
- Functions that return a single value
- Literal values
- Other 4GL expressions

Named Values Used as Operands

A named value is a variable of any simple data type, an element of an array, a member of a record or the constants TRUE, FALSE, NOTFOUND:

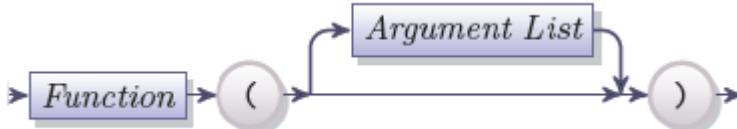


Element	Description
Array	The name of a variable of either ARRAY or DYNAMIC ARRAY data type
Record	The name of a variable of the RECORD data type
Variable	The name of a variable of simple data type
Constant	one of the built-in constants: TRUE, FALSE or NOTFOUND
Index	comma separated list of up to three integer expressions specifying the index of an array element

An identifier of a variable of TEXT or BYTE data type can be used as an operand in a 4GL expression only together with IS NULL, IS NOT NULL and WORDWRAP operators.

Functions Used as Operands

Calls for functions that return a single value can be used as operands of 4GL expressions.



Element	Description
Function	The name of a function
Argument List	Optional list of arguments passed to the function, if it is empty, the parentheses must still be present

It can be a built-in function or a programmer-defined function, provided that it returns only one value of the data type that is valid in the 4GL expression in which the function operand is used.

Literal Values Used as Operands

Literal values can be used as operands. A literal value can be either a number or it can be a character string. Character strings used as operands must be enclosed in double quotation marks ("..."). Digits must be enclosed in quotation marks only if they represent DATETIME, DATE, INTERVAL values in the form of a character string, in other cases the usage of the quotation marks is optional.

Other 4GL Expressions Used as Operands

Two or more expressions can occur as operands within one expression only if they are used with separators. You cannot use too complex nested expressions; if an expression is too complex, an error will appear. You should substitute one complex expression by two or more simple expressions. When an expression returns a

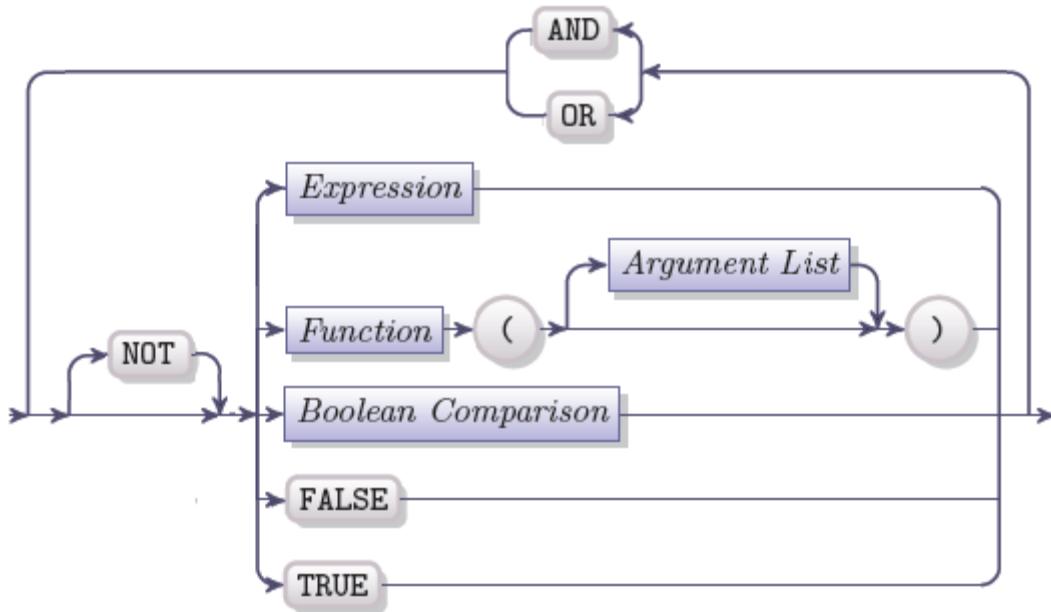


different data type from which is expected, Querix 4GL will try to convert this value to the required data type.



Boolean Expressions

A Boolean expression is a particular case of an Integer expression (which in its turn is a particular case of Numeric expressions) that returns either TRUE or FALSE. TRUE has the numeric value of 1, FALSE is equal to 0. In some cases, a Boolean expression can return NULL. The syntax of a Querix 4GL Boolean expression is identical to that of the SQL.



Element	Description
Expression	A 4GL expression
Boolean Comparison	An expression that tests operands against the specified condition and returns a Boolean value
Function	The name of a function that returns a single value
Argument list	Optional arguments passed to the function

There are so called Boolean operators and Boolean comparison that are used in the Boolean expressions. Boolean operators are the logical operators AND, OR, NOT, they are used to combine Boolean values into a Boolean expression.



Boolean comparison is used to test the operands and to return Boolean values.

Testing for	Operators used	Example	
		TRUE	FALSE
Equality or inequality of values	Relational operators: == or =, <> or !=, <, >, <=, >=	10<=100 "Tom" = "John"	
Equality or inequality of character strings	MATCHES and LIKE	IF "string" MATCHES "*rin*" IF "string" LIKE "%nir%"	
Null values	IS NULL and IS NOT NULL		IF text_var IS NOT NULL
Correspondence of a value to a range	BETWEEN ... AND		IF 10 NOT BETWEEN 5 AND 15
Membership of a value	IN()	IF 4 NOT IN(1,2,3)	IF 10 IN(6,7,8)

A Boolean expression can belong at the same time to any of the other types of the 4GL expressions. To store the value returned by a Boolean expression INT and SMALLINT data types can be used. However, if you compare two operands that have different data types, unexpected results can occur. Numbers are generally compared with numbers, time values with time values and character strings with character strings.

	Note: A period of time represented by a value of INTERVAL data type cannot be compared to a moment in time represented by a value of DATETIME or DATE data types. A variable or expression to which an INTERVAL value is compared must return value of the INTERVAL data type.
--	---

Evaluation of the Boolean Expressions

After a Boolean expression is evaluated, Querix 4GL evaluates the expression as TRUE, FALSE or NULL depending on the returned value and on the context in which the Boolean expression is used.

A Boolean expression is evaluated as TRUE if it returns:

- A non-zero real number
- A character string that consists of a non-zero real number
- A non-zero value of INTERVAL data type
- Any DATE or DATETIME value
- A TRUE value returned by a function of infield() type
- The built-in integer constant TRUE

	Note: A Boolean expression is evaluated as TRUE also if an expression that is an operand of the IS NULL operator returns NULL.
--	---

A Boolean expression is evaluated as NULL if it returns NULL and is **not** used in:

- A NULL test
- A Boolean comparison
- A 4GL conditional statement (WHILE, IF, CASE)



 A circular icon with a black border and a purple gradient background. Inside is a white circle containing a black lowercase letter 'i'.	Note: If one or more elements of the comparison returns NULL, all the Boolean expression used in a 4GL conditional statement is evaluated as FALSE.
---	--

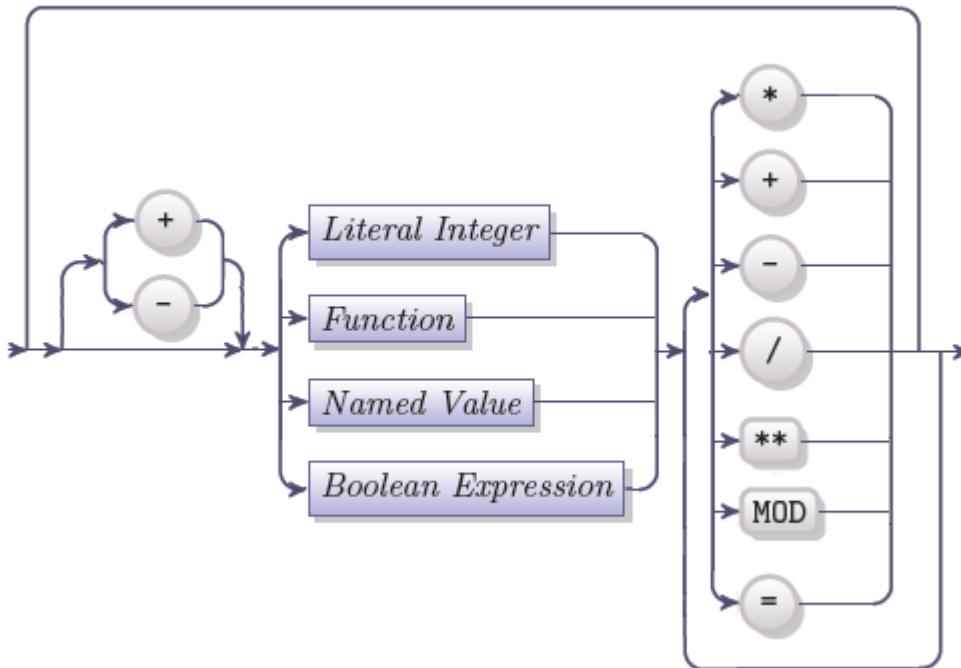
In all the other cases a Boolean expression is evaluated as FALSE.

If a Boolean expression is put in the context where a numeric value is expected, the returned value will be converted to an integer (TRUE will be converted to 1, FALSE will be converted to 0).



Integer Expressions

An Integer expression is a particular case of a Numeric expression that returns whole number.



Element	Description
Literal Integer	The base-10 representation of an whole number
Function	The function call of a function which returns exactly one value
Named Value	A variable of integer or compatible data type, a member of a record or an element of an array
Boolean Expression	An expression which returns either TRUE or FALSE

Function calls can be used in an Integer expression only if they return values of INTEGER data type. Values of the DATE data type can be used in the Integer expressions when they are used to define the difference between two DATE values, because the difference is returned in the format of an integer.

	Note: The returned difference between two values of DATE data type has INTEGER data type, to convert it to the INTERVAL data type use UNITS operator.
--	--

Integer expressions can be used as operands within any other 4GL expression. An Integer expression is a logical subset of Numeric expressions. They are described separately because some 4GL operators, built-in functions, statements and form specifications are valid only when used with the values of INTEGER data type, and some of them can be used only with positive integers.



Unary Arithmetic Operations

The unary operators that can be used with integers are plus (+) and minus (-) signs that can be placed to the left of an Integer expression or its component to indicate its sign. If a unary operator is absent the default positive operator (+) is assigned to it implicitly. If you need to use a subtraction operator in close succession with a negative unary operator (-), use parentheses to separate the negative number with its sign from the subtraction operator.

E.g.: $25 - (-6)$

Otherwise two successive negative signs will result in a comment operator (--). The unary positive and negative operators are recursive.

Binary Arithmetic Operations

The following binary arithmetic operators can be used in Integer expressions with either left-hand or right-hand operands:

Operator	Precedence	Action	Result
**	13	Exponentiation	Power
MOD		Modulus	Integer remainder
*	12	Multiplication	Product
/		Division	Quotient
+	11	Addition	Sum
-		Subtraction	Difference

All the arithmetic operations are performed after both operands are converted to the DECIMAL data type. MOD operands are converted from the INTEGER data type and then to the DECIMAL data type.

If there are two or more operators with the same precedence within one expression, they are processed from left to right. An arithmetic expression containing at least one NULL value returns NULL.

An Integer expression that specifies the right-hand MODE operator or an array element cannot contain the exponentiation operator or the modulus operator, it also cannot return zero. The exponentiation operator cannot possess a right-hand integer operand that has negative value.

	Note: A word 'mod' cannot be used as a 4GL identifier as it is already reserved for the modulus operator.
---	--

The results of the division action where both operands are of INT or SMALLINT data type will the quotient with discarded fractional part. If you try to divide by zero (that is if the right-hand operand evaluates to zero), an error will occur.

The above listed arithmetic operators can be applied to Numeric expressions and to Time expressions, though some restrictions will be applied.



Literal Integers

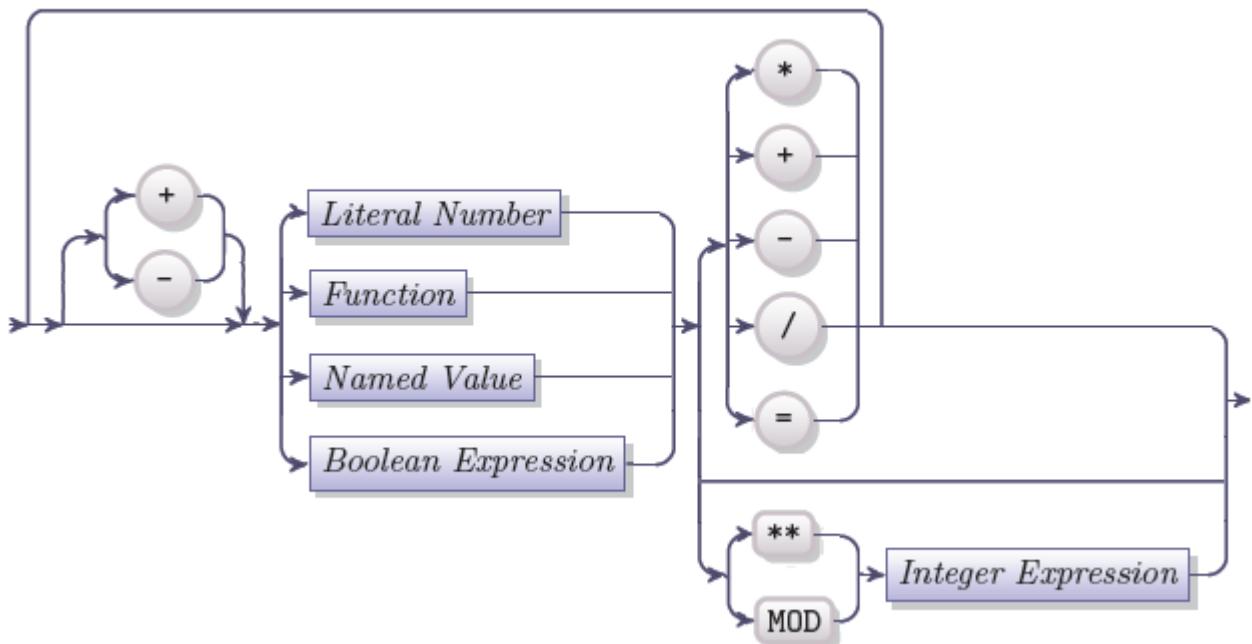
A literal integer is a numeric representation of the INTEGER data type, entered in the decimal number system format **without** whitespaces, commas or a decimal point. Non-ASCII digits are not recognized in the Numeric expressions on the whole and in Integer expressions in particular. The literal integers can have a unary plus or minus sign on the left. If it is omitted a literal integer is treated as a positive integer by default.

E.g.: 12345 -678 +90



Numeric Expressions

A Numeric expression is evaluated to a real number; it includes Boolean expressions and Integer expressions discussed above. They also can include literal numbers and function calls, they should return a single value of DECIMAL, FLOAT, INTEGER, MONEY, SMALLFLOAT, or SMALLINT data type to be a valid operand of a Numeric expression.



Element	Description
Literal Number	The base-10 representation of a real number
Function	The function call of a function which returns exactly one value
Named Value	A variable of numeric data type or other compatible data type, a member of a record or an element of an array
Boolean Expression	An expression which returns either TRUE or FALSE
Integer Expression	An expression which returns a positive or negative integer

The whole expression evaluates to NULL if at least one of its operands evaluates to NULL. The range of values that an expression can return is determined by the receiving data type.

Arithmetic Operators

As Numeric expressions include also Integer expressions, the [binary arithmetic operators](#) and [unary arithmetic operations](#) described in the Integer expressions section also apply to Numeric expressions, though the following restrictions are applied:

Any modulus or right-hand exponentiation operator is converted first to INTEGER data type and then to DECIMAL data type. This may result in discarding fractional parts of the operands.



A Numeric expression will return a whole number, if both of the operands are of INTEGER, SMALLINT or DATE data type. If either operand of a Numeric expression is of DECIMAL, FLOAT, MONEY, or SMALLFLOAT data type, the returned value may have fractional part unless the MOD operator is used.

Literal Numbers

A literal number is a numeric representation of any numeric data type, entered in the decimal number system format as a fixed-point decimal number or in exponential notation. A literal number cannot contain comma symbol (,) or whitespaces (ASCII 32). A unary sign (+ or -) can precede either mantissa or exponent or both. Querix 4GL does not support non-ASCII digits used in literal numbers.

Three kinds of literal numbers can be used in Querix 4GL:

- Integer literals of INTEGER and SMALLINT data types, they do not have a decimal point.
-123 4567 +890
- Fixed-point decimal number values of DECIMAL(p,s) and MONEY data types, they can include decimal point
12345.67 -12345.0
- Floating-point decimal number values of FLOAT, SMALLFLOAT, and DECIMAL(p)
12.345E6 -12345.6E7 -1234567.0E4

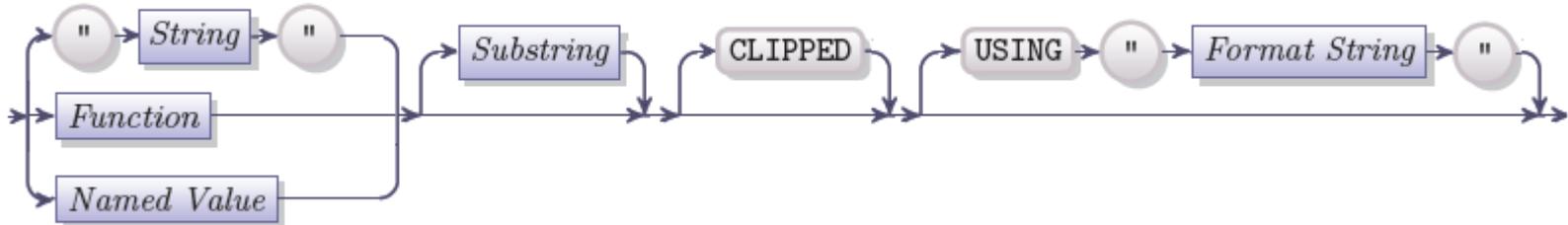
A literal number that represents a value of MONEY data type should not be preceded with a currency symbol. Currency symbols are displayed next to the MONEY values automatically. By default a dollar symbol (\$) is displayed, this setting can be overridden with the settings specified in the DBMONEY or DBFORMAT environment variable.

If the data type of a literal number is different from what is expected, Querix 4GL performs conversion if possible. You can still receive unpredictable results if a literal number within a Boolean expression is of a data type that cannot represent the data type with which the Boolean expression is compared. E.g.: Relational operators cannot return TRUE when one of the operands returns a floating-point number and the other is of INTEGER data type because of the rounding error that may occur. Unexpected results will also occur if you try to use a binary number where a decimal number is required.



Character Expressions

A character expression evaluates to a character string. The maximum length of the character string is the same as the maximum length of the CHAR data type (32,767 bytes) of VARCHAR data type (unlimited).



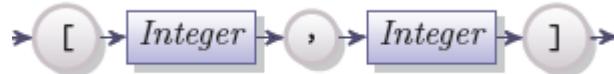
Element	Description
String	a string of characters
Function	The function call of a function which returns exactly one value
Named Value	A variable of character or compatible data type, a member of a record or an element of an array
Substring	Optional coordinate of the substring within a character value
Format String	a formatting mask to specify how 4GL displays the returned character value

A Character expression cannot contain a value of TEXT data type (except in the NULL test or as a WORDWRAP operand within a PRINT statement).

If a Character expression contains a value of a data type different from CHAR, STRING or VARCHAR, a data type conversion will take place.

Character Substrings

Two integers enclosed into the square brackets that follow a Character expression specify a character substring:



Element	Description
Integer	The integer specifying the position of the first or the last character of the substring character sequence

The first and the second numbers indicate the first and the last character within a value returned by a Character expression (respectively). The last number cannot be larger than the declared size of the receiving data type (or the length of the string itself) and the first number should be smaller than the second one.

```
cl_city[1,7]
```

The above character string specifies the first seven characters of the 'cl-city' variable.



A character substring can be specified for an array of CHAR, STRING or VARCHAR data type.

```
array1[4][1,7]
```

The above character substring specifies the first seven characters within an array element located on the fourth row. No exponentiation or modulus operators can occur within the integer expressions that specify an array element or a substring. Other arithmetic operators and parentheses can be used.

String Operators

A format of the resulting character string can be changed with the help of USING keyword. Forms and reports possess a wider range of formatting tools for text values. A value of CHAR data type can contain trailing blanks which may be unwanted, to discard them use CLIPPED operator, it can be used with the expression on the whole or with its components.

SPACE and COLUMN operators are used to change the format of values in DISPLAY or PRINT statements. They add whitespaces and empty columns. The SPACES keyword can be used as a synonym for SPACE keyword.

Printable and Non-Printable Characters

Querix 4GL regards the following characters as **printable** by default (in the U.S. English locale):

- ASCII 32 (= blank) through ASCII 126 (= ~)
- FORMFEED (= CONTROL-L)
- NEWLINE (= CONTROL-J)
- TAB (= CONTROL-I)

Any other characters are regarded as **non-printable**. Though character strings containing non-printable characters can be used as operands as well as they can be returned by a Character expression and stored in CHAR, STRING, VARCHAR and TEXT database columns, some manipulations are not available for non-printable characters.

The following problems can occur if CHAR, STRING, VARCHAR or TEXT values include non-printable characters:

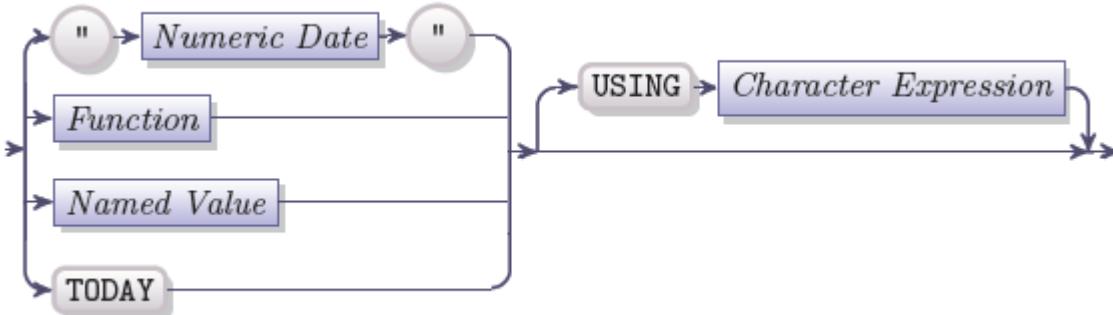
- Input and output behaviour for WORDWRAP attribute and DISPLAY and PRINT statements is designed for work with printable characters only. Unpredictable and unpleasant results may occur, if they are used with non-printable characters.
- Output of character strings that contain non-printable characters to a peripheral device can also cause unpredictable results. Some sets of non-printable characters can misplace a cursor, modify the terminal attributes or make the screen unreadable.
- The non-printable characters CONTROL-D (= ASCII 4) and CONTROL-Z (= ASCII 26) can be treated as logical end-of-file in output from a report.
- A zero byte (ASCII 0) stored in a variable of CHAR, STRING or VARCHAR data type can be treated either as string terminator or as data depending on the operators used.

To avoid the above mentioned problems use the BYTE data type, if you need to store non-printable characters. The list of non-printable characters can differ from the given in this chapter due to usage of non-default locales. The DBAPICODE environment variable can be used to specify the character-mapping file of the character set of a peripheral device and a database.



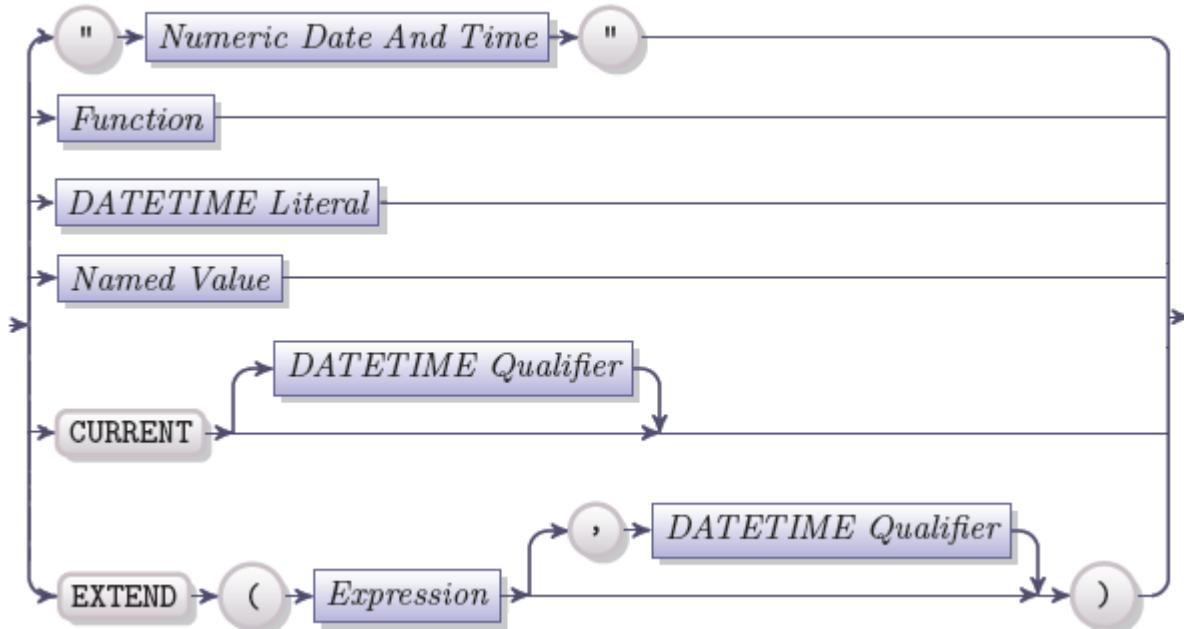
Time Expressions

A Time expressions evaluates to a value of the time data type (DATE, DATETIME or INTERVAL). The DATE expression returns the DATE value has the following syntax:



Element	Description
Numeric Date	The date represented by digits and separator signs
Function	The function call of a function which returns exactly one value
Named Value	A variable, a member of a record or an element of an array of the DATE data type or compatible
Character Expression	A character expression containing the format string

The DATETIME expression returns a DATETIME value and has the following syntax:

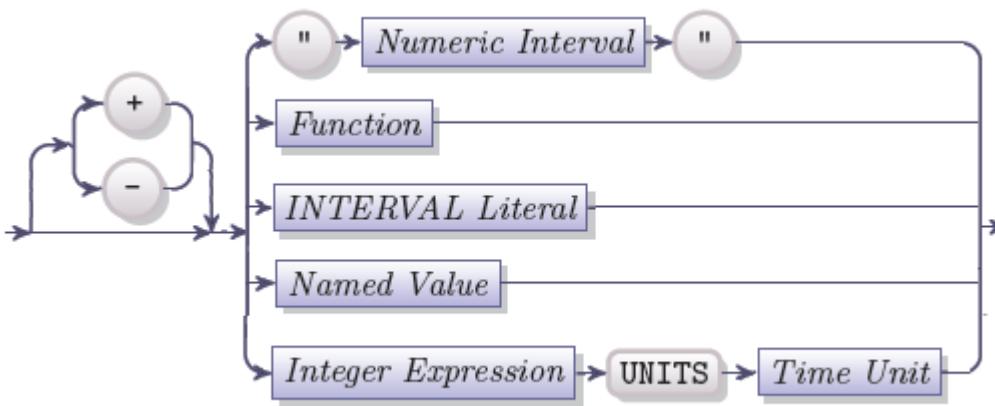


Element	Description
---------	-------------



Numeric Date And Time	The date and time value represented by digits and separator signs
Function	The function call of a function which returns exactly one value
DATETIME Literal	A numeric representation of a DATETIME value with separators followed by the DATETIME qualifier
Named Value	A variable, a member of a record or an element of an array of the DATETIME data type or compatible
Expression	A DATE or DATETIME expression
DATETIME Qualifier	The range of the time units from the largest to the smallest

The INTERVAL expression returns a value of the INTERVAL data type and has the following syntax:



Element	Description
Numeric Interval	The interval value represented by digits and separator signs
Function	The function call of a function which returns exactly one value
INTERVAL Literal	A numeric representation of an INTERVAL value with separators followed by the INTERVAL qualifier
Named Value	A variable, a member of a record or an element of an array of the INTERVAL data type or compatible
Integer Expression	An expression returning a positive or negative integer
Time Unit	A keyword specifying the time unit: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION

DATE and DATETIME operands may seem similar, though they are treated differently in arithmetic operations, FORMAT and PICTURE attributes influence them differently, and their values are stored in different data types.

All the three data types belong to time data types. However, conversion between them is very limited. DATE and DATETIME values can be interchangeable in the context where a time expression is required. The values of INTERVAL data type cannot be converted to DATE or DATETIME data type.

Numeric date in DATE Expressions

A numeric date is a string of digits and corresponding delimiters enclosed into double or single quotation marks. It should have the following format:



“mm/dd/yy” or “mm/dd/yyyy”

where mm is months, dd is days, yy and yyyy is year, this is the default order and format of the time units. The default delimiter is a slash (/). The delimiters and the time unit format can be changed with the help of the DBDATE environment variable. This can also be done by means of USING operator or the FORMAT attribute. The environment variable GL_DATE offers wider opportunities for formatting a date value.

The DBCENTURY environment variable stores the directions according to which Querix 4GL expands the year value if not all the four digits have been entered for a year value.

If the quotation marks are omitted, Querix 4GL makes an attempt to treat the entered value as a literal integer. The slash separator is treated as a division symbol. The quotient of such operation is usually equal to zero or December 31, 1899. It is advisable not to omit the quotation marks in the numeric date specification.

Numeric Values and Literals in DATETIME and INTERVAL Expressions

A literal of either DATETIME or INTERVAL data type consists of the numeric representation of the corresponding value enclosed into parentheses and with corresponding delimiters followed by the DATETIME qualifier or INTERVAL qualifier (respectively).

The numeric values (“numeric date and time” for DATETIME expressions and “numeric interval” for INTERVAL expressions) can be used either as independent operands of the corresponding expressions or like component parts of the corresponding literals. Numeric values, qualifiers and literals of these data types are discussed in this manual in the chapter devoted to Querix 4GL data types. See the “[DATETIME Qualifiers](#)” and “[INTERVAL Qualifiers](#)” section of the first chapter of this manual for more information about qualifiers. See the “[DATETIME Literal](#)” and “[INTERVAL Literal](#)” sections for information about literals. For information about the numeric time values see the “[Numeric Interval Value](#)” and “[Numeric Date and Time Value](#)” sections.

Arithmetic operations in Time Expressions

According to the three types of Time expressions they can return values of three data types. Still there are rules which apply to all the Time expressions:

- If the first operand of an arithmetic expression contains UNITS keyword, it should be enclosed in parentheses.
- If any of the operands of a Time expression evaluates to NULL, the whole expression will return NULL

Arithmetic Operations on DATE Expressions

Values of DATE data type are stored as integers that specify the number of days that have passed since December 31, 1899. Thus DATE expressions can be used together with Integer expressions in arithmetic operations, where an integer will represent a number of days added to a DATE value or subtracted from it. There are only two operators that can be used within this type of Time expressions:

First Operand	Operator	Second Operand	Result
---------------	----------	----------------	--------



DATE Expression	+ (Addition)	Integer Expression	A DATE value increased by the number of days specified in the Integer expression
Integer Expression		DATE Expression	
DATE Expression	- (Subtraction)	Integer Expression	A DATE value decreased by the number of days specified in the Integer expression
Integer Expression		DATE Expression	
DATE Expression		DATE Expression	An integer number that specifies the number of days that elapsed between the two dates

You can define the difference between two DATE values using minus (-) operator, the result will be a positive or negative INTERVAL value that represents the number of days that elapsed between the two dates. However, to use the result of the operation (DATE expression – DATE expression) as an INTERVAL value, you need to apply the UNITS DAY operator to the resulting value.

DATE expressions support addition, subtraction and relational operators operations, but they do not support multiplication, division, and exponentiation operators.

Arithmetic operations on DATETIME Expressions

Since DATETIME values are stored as fixed-point decimal numbers, some arithmetic operations can be performed on them. Like with DATE data type there are only two arithmetic operators that can be used with the DATETIME expressions.

First Operand	Operator	Second Operand	Result
DATETIME Expression	+ (Addition)	INTERVAL Expression	A DATETIME value increased by value of INTERVAL expression
DATE Expression		DATETIME Expression	
INTERVAL Expression		DATE Expression	
DATETIME Expression		INTERVAL Expression	
DATE Expression	- (Subtraction)	INTERVAL Expression	A DATETIME value decreased by value of INTERVAL expression



	Note: Other combinations or operands are not allowed. E.g. you cannot subtract DATE or DATETIME expression from an INTERVAL expression, you cannot also add two DATE or DATETIME values.
---	---

Arithmetic operations on INTERVAL Expressions

Since INTERVAL values are stored as DECIMAL data type, the following operators can be used with INTERVAL expressions:

First Operand	Operator	Second Operand	Result
INTERVAL Expression	+ (Addition)	INTERVAL Expression	An INTERVAL value increased by the value of the second operand
		Number Expression	
INTERVAL Expression	* (Multiplication)	INTERVAL Expression	An INTERVAL value multiplied by the value of the second operand
		Number Expression	An INTERVAL value multiplied by the value of the second operand
INTERVAL Expression	/ (Division)	INTERVAL Expression	An INTERVAL value divided by the value of the second operand
		Number Expression	
INTERVAL Expression	- (Subtraction)	INTERVAL Expression	An INTERVAL value decreased by the value of the second operand
DATE Expression		DATETIME Expression	An INTERVAL value that represents the time elapsed between the moments specified by DATE and DATETIME values
DATETIME Expression		DATE Expression	
		DATETIME Expression	

One and the same Time expression **cannot** contain INTERVAL expressions of two different types: YEAR TO MONTH and DAY TO FRACTION INTERVAL. The DATE and DATETIME values can be used in the arithmetic operations with INTERVAL if it has the corresponding qualifiers. EXTEND keyword can be used to make DATE and DATETIME values match the qualifier of an INTERVAL value.

Numeric values for DATETIME and INTERVAL data types ("numeric date and time" and "numeric interval") cannot be used in arithmetic expressions, literal values should be used instead.

In some cases, arithmetic operations with the use of UNITS operator or operands of INTERVAL data type may return invalid dates, 4GL will produce the error 1267 "Precision overflow on DATETIME or INTERVAL operation". E.g. If you try to execute the following line

```
(10 UNITS DAY) + DATETIME(2010-08-27)
```



The returned value will not be a valid date, because the $27+10=37$ days which is not possible.

Relational Operators Used with Time Expressions

Relational operators that can be applied to Time values obey to such rules:

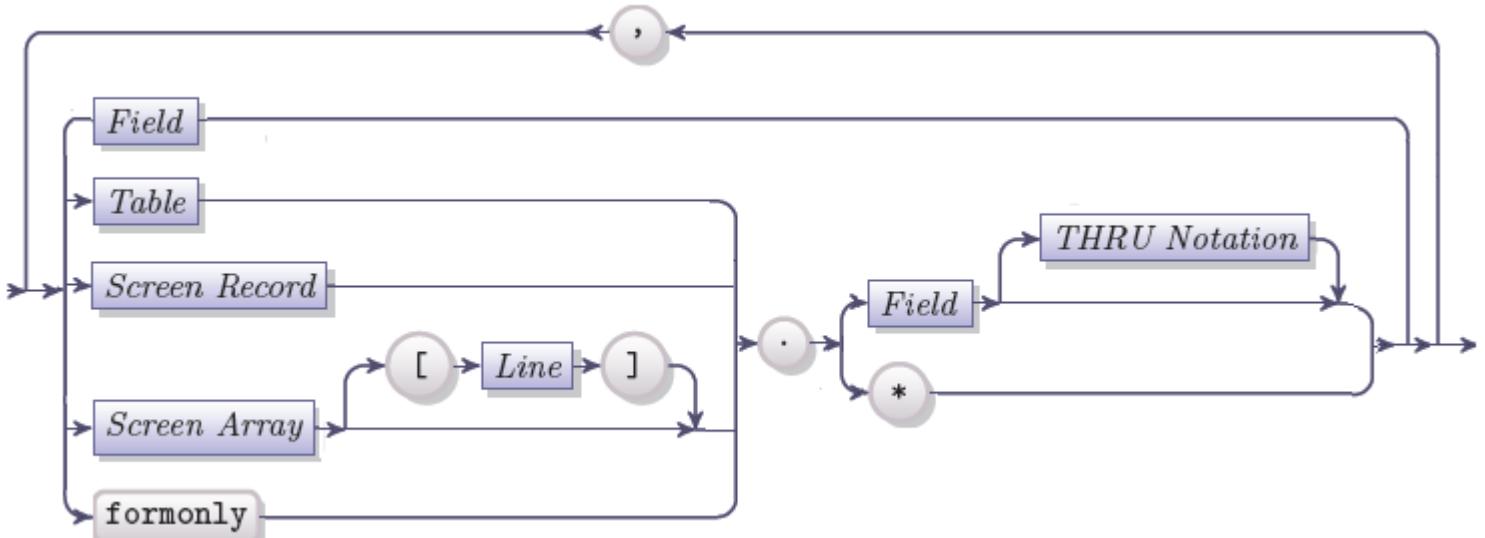
First Operand	Operator	Second Operand	Evaluates to TRUE when:
INTERVAL Expression	< (Less than)	INTERVAL Expression	first operand represents a briefer period of time
DATE Expression or DATETIME Expression		DATE Expression or DATETIME Expression	first operand represents an earlier moment in time
INTERVAL Expression	> (Greater than)	INTERVAL Expression	first operand represents a longer period of time
DATE Expression or DATETIME Expression		DATE Expression or DATETIME Expression	first operand represents an later moment in time

INTERVAL operands cannot be mixed with DATE or DATETIME operands when a relational operand is used.



Field Clauses

A field clause is used to specify one or more form fields or screen records.



Element	Description
Table	The name of a table or a view to which the field is linked
Screen Record	The name of a screen record declared in a form file
Screen Array	The name of the screen array specified in a form file
Line	An integer expression, enclosed within brackets, to specify a record within the screen array. Here $1 \leq \text{line} \leq \text{size}$, where size is the array size that is declared in the INSTRUCTIONS section. If it is omitted, the first record of the array will be used
Field	The name of the form field as declared in the ATTRIBUTES section of a form file
THRU Notation	The THRU or TROUGH keyword followed by the last field of the field sequence

A reference to a database table must not include table qualifiers. An alias should be declared in the TABLES section of a form specification file for a table that requires a table qualifier (server, owner, etc.).

An asterisk (*) can be used after the dot to specify all the fields in the specified screen record, screen array, table or to specify all the fields that are not associated with the table columns (*formonly.**). You can find the information about the THRU/THROUGH keyword in the "[Asterisk\(*\) and THRU/THROUGH Keyword](#)" section of this chapter.

In some contexts, only a field clause with one field identifier is supported (e.g. NEXT FIELD clause, PREVIOUS FIELD clause, etc.).



Note: Some Querix 4GL statements support only some of the features specified in the syntax diagram. E.g.: Only the first record of a screen array can be specified in a CONSTRUCT statement

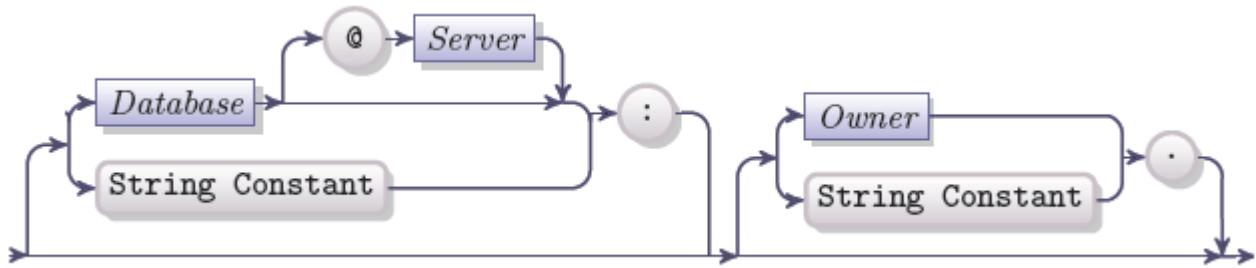
The following examples show the cases where a field clause can be used:

```
INPUT BY NAME order_num, order_qty, order_client
CLEAR order_form[10].*
CLEAR lname, fname, age, address_rec.*
SCROLL my_order.* UP 10
```



Table Qualifiers

Table qualifiers can appear in the SQL of 4GL statements, as well as in the declaration of a table alias within the TABLES section of a form specification file. However, table qualifiers cannot be included in other parts of a form specification file, table aliases are needed instead. It is also not possible to prefix a form field or an alias with a table qualifier.



Element	Description
Database	The name of a database which contains the table, view, or synonym
Server	The name of the server on which the database is running
Owner	The login name of the owner of the table, view, or synonym. It immediately precedes the table name
String Constant	The schema ID or the owner qualifier written as quoted character strings for ANSI compatibility

A complete table qualifier has the following structure:

database@server:owner

If you use the ANSI-compliant quoted character strings, it can look like the following:

“database@server”：“owner”

The full name of a table column with a complete table qualifier will look like follows:

database@server:owner.table.column

Owner Qualifier

It is obligatory to specify the owner of a table in a table qualifier if there is another table with the same name within the same database. In an ANSI-compliant database you must specify the owner of all the tables which you do not own, even if they have unique names. If you use an ANSI-compliant database as your current database, an error will occur if you try to reference a not ANSI-compliant database as a remote database.

You are free to include the owner of a table in a not ANSI-compliant database which has a unique identifier; however, if the owner qualifier is incorrect then the 4GL will produce an error.



Referencing Database

Either *database:* or *database@server:* qualifier can be used with the LIKE keyword (within the DEFINE, INITIALIZE, VALIDATE statements) to specify a table or a table column that is not in the current database. The tables and table columns without such qualifier will be searched in the current database, which needs to be specified before the first program block within the same module. To be able to use the LIKE keyword within the above mentioned statements, you need to include the DATABASE statement that will specify the current database.

	Note: Even if you specify a non-current database qualifier in the LIKE clause, you still need to specify the current database otherwise 4GL will produce an error.
---	---

The current database is specified by the most recently used DATABASE statement, it can be used repeatedly within a function or the main block. The only action that can be done with the information in a database, which is not your current database and which is referenced with the help of a table qualifier, is to use the SELECT statement. It is not possible to modify (delete, update, or insert) rows within a database that is not your current database.

The qualifier *@server* is optional and is used after the *database* name in a table qualifier, if the table resides on another host system, e.g.:

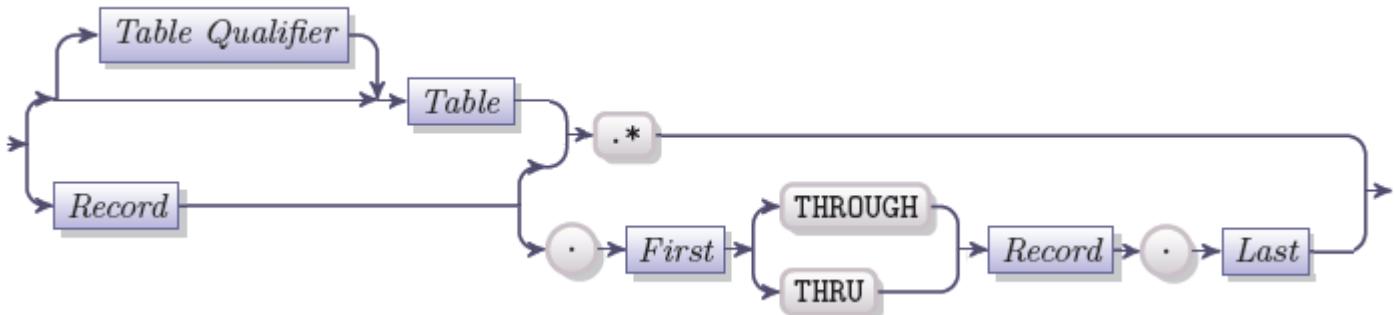
```
cms@my_host:smith_john.order_table
```

If a database resides in your current directory, or if the location of a database is specified in DBPATH environment variable, you do not need to specify the server name.



Asterisk (*) and THRU/THROUGH Keyword

To specify a sub-list of successive array elements or record members the THRU (synonym is THROUGH) keyword may be used. To specify all the array elements, record members or all the columns within a table in a consecutive order, the (.*.) notation is used.



Element	Description
Table Qualifier	Optional qualifiers of the table, such as database, server, owner. For more information see “ Table Qualifiers ” section above
Record	A screen or program record. It should be one and the same record in both cases
Table	The name of a database table
First	The name of a program record member or the name of a field which belongs to the screen record
Last	The name of a program record member or the name of a field which belongs to the screen record. It must be declared later than the first

	Note: Table columns cannot be referenced with the help of the THRU/THROUGH keyword.
--	--

```

INPUT my_scr_record.* FROM order_table.*
DISPLAY my_rec.fname THRU my_rec.address TO screen_rec.*

```

When the THRU keyword is used with a record it has the format *record.first THRU record.last*. The *first* is the first member of a subset of the record members you want to specify, it is not necessarily the first member of the record. The *last* is the last member of this subset, it is not necessarily the last member of the record. They must belong to the same record. This notation specifies all the record members between the first member and the last member of the subset (inclusively). The *first* should appear earlier than the *last* in the list of the record members in the declaration of a program record. The *first* should also appear earlier than the *last* in the list of fields in the screen record specification within the INSTRUCTIONS section of a form specification file.



The discussed notations are usually used to substitute complete or partial lists of members, elements, fields or columns. The pairs of the examples below will produce identical results:

```
DISPLAY program_rec.* TO screen_rec.*  
DISPLAY program_rec.first_r, program_rec.second_r, program_rec.third_r,  
program_rec.last_r TO screen_rec.first_fld, screen_rec.second_fld,  
screen_rec.third_fld, screen_rec.last_fld
```

and

```
INITIALIZE program_rec.second_r THRU screen_rec.last_r TO NULL  
INITIALIZE program_rec.second_r, program_rec.third_r, program_rec.last_r  
TO NULL
```

The examples with the notations are compact and they help to save time and efforts of a developer. Such notations help to avoid mistakes which tend to occur when a large amount of information is rewritten many times.

The order of the fields in a screen record depends on their order in the INSTRUCTIONS section within a form specification file. They can be listed in any order, but this order will be fixed when this screen record will be used in a program:

```
INSTRUCTIONS  
SCREEN RECORD screen_rec (  
    formonly.f003,  
    formonly.f001,  
    formonly.f002,  
    formonly.f000,  
    formonly.f004  
)
```

It is possible to use the THRU keyword in the INSTRUCTIONS section and not to rewrite a list of the form fields. In this case the fields in a screen record will be ordered according to their physical order in the ATTRIBUTES section of a form specification file:

```
ATTRIBUTES  
f000 = formonly.f000;  
f001 = formonly.f001;  
f002 = formonly.f002;  
f003 = formonly.f003;  
f004 = formonly.f004;  
INSTRUCTIONS  
SCREEN RECORD screen_rec (formonly.f000 THRU formonly.f004)
```



The physical position of the fields in the SCREEN section of a form specification file is ignored. Querix 4GL also ignores the order of the table names in the TABLES section, the lexicographical order of the filed names and the CONSTRAINED and UNCONSTRAINED keywords used in the OPTIONS statement.

There are cases when neither THRU keyword (.* notation can be used to represent a list of fields, columns, members or elements:

- THRU or THROUGH keyword cannot be used for
 - referencing a partial list of table columns;
 - referencing a partial list of the members of a screen record if a program is displaying the data to the form or the input in the form is performed;
 - specifying variables of the SELECT or INSERT clause within a PREPARE statement
 - referencing a program record that contains an array element (still it can be used to reference records that contain other records as members and records that are elements of arrays)
- (.* notation cannot be used for
 - specifying variables of the SELECT or INSERT clause within a PREPARE statement
 - referencing a program record that contains an array element (still it can be used to reference records that contain other records as members and records that are elements of arrays)



Note: When the (.* notation that specifies all the columns of a table is used in UPDATE statement, the column of SERIAL data type is omitted and is not included in the list referenced by the notation *table.**



ATTRIBUTE clause

Some Querix 4GL statements support different display and control attributes. To assign them the ATTRIBUTE clause must be used. The general syntax of such a clause is as follows:



All of the attributes specified in the attribute clause in the Querix 4GL statements must be enclosed in parentheses and separated by commas. Otherwise a compilation error occurs.

If the ATTRIBUTE clause is included into a statement, it must contain at least one attribute.

The statements which may require the attribute clause and the attributes supported by them are listed in the table below.

	Statement	Attributes
CONSTRUCT	display	✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE, NORMAL ✓ REVERSE, UNDERLINE
	control	✓ NAME, HELP, FIELD ORDER FORM, ACCEPT, CANCEL
	control	✓ FIELD ORDER FORM, UNBUFFERED
DIALOG		
INPUT	control	✓ NAME, HELP, WITHOUT DEFAULTS
INPUT ARRAY	control	✓ APPEND ROW, INSERT ROW, DELETE ROW, AUTO APPEND, KEEP CURRENT ROW, HELP, COUNT, MAXCOUNT, WITHOUT DEFAULTS
CONSTRUCT	control	✓ NAME, HELP
DISPLAY ARRAY	control	✓ COUNT, HELP
DISPLAY	display	✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE ✓ REVERSE, UNDERLINE
	control	✓ COUNT, HELP, KEEP CURRENT ROW, UNBUFFERED, ACCEPT, CANCEL
DISPLAY AT	display	✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE, NORMAL ✓ REVERSE, UNDERLINE



DISPLAY BY NAME	
DISPLAY FORM	
DISPLAY TO	
ERROR	display <ul style="list-style-type: none"> ✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE ✓ REVERSE, UNDERLINE ✓ STYLE
MESSAGE	
INPUT	display <ul style="list-style-type: none"> ✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE, NORMAL ✓ REVERSE, UNDERLINE control <ul style="list-style-type: none"> ✓ ACCEPT, CANCEL, FIELD ORDER FORM, HELP, NAME, UNBUFFERED, WITHOUT DEFAULTS
INPUT ARRAY	display <ul style="list-style-type: none"> ✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE, NORMAL ✓ REVERSE, UNDERLINE control <ul style="list-style-type: none"> ✓ ACCEPT, CANCEL, APPEND ROW, INSERT ROW, DELETE ROW, AUTO APPEND, KEEP CURRENT ROW, HELP, COUNT, MAXCOUNT, FIELD ORDER FORM, UNBUFFERED, WITHOUT DEFAULTS ✓ STYLE, COMMENT, IMAGE
MENU	control
OPEN WINDOW	window <ul style="list-style-type: none"> ✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE ✓ REVERSE, UNDERLINE ✓ TEXT ✓ STYLE ✓ ERROR LINE, COMMENT LINE, FORM LINE, MENU LINE, MESSAGE LINE, PROMPT LINE ✓ BORDER
OPTIONS	display <ul style="list-style-type: none"> ✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE, NORMAL ✓ REVERSE, UNDERLINE
PROMPT	display <ul style="list-style-type: none"> ✓ BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, YELLOW ✓ BOLD, DIM, INVISIBLE, NORMAL ✓ REVERSE, UNDERLINE control <ul style="list-style-type: none"> ✓ ACCEPT, CANCEL, CENTURY, FORMAT, PICTURE, SHIFT, WITHOUT DEFAULTS

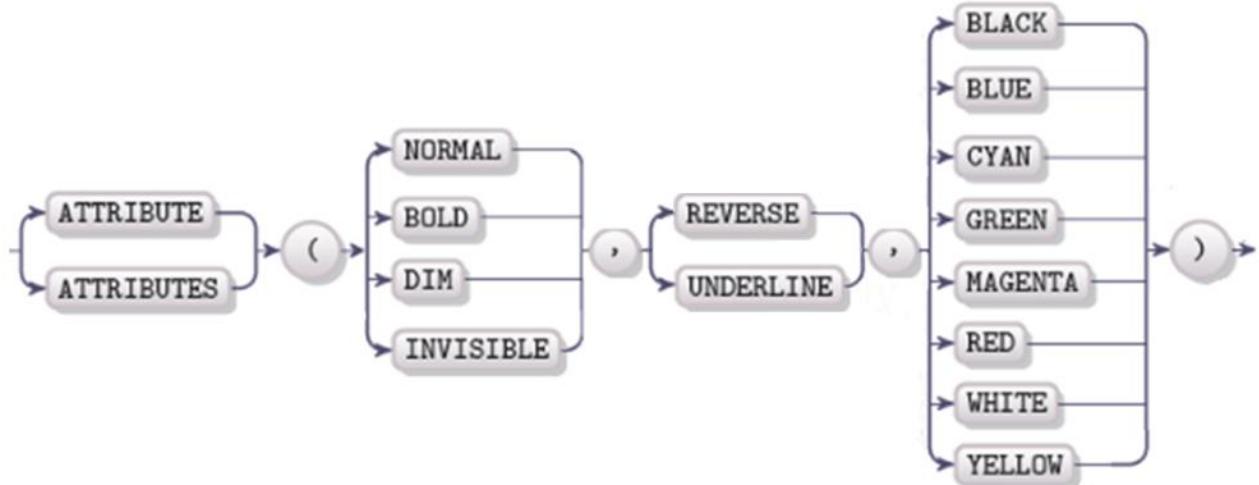
As you can see from the table some of the statements support only display or only control attributes, and there are statements which may require attribute usage of both classes. Besides some of them support a short and specific number of attributes. As this chapter is dedicated to the general description of the possible attributes and specification of the commonly used attributes, to find more detailed information about the targeted statement you should proceed to the section describing its usage.



Display Attributes

Display attributes are mainly used to define the colour, font intensity and view of the information displayed.

The ATTRIBUTE clause with the display attributes has the following syntax:



As it was mentioned in the general attribute clause description when used the attributes must be enclosed in parentheses and separated by commas.

All display attributes can be divided into 3 attribute types:

- the font intensity attributes: BOLD, DIM, NORMAL and INVISIBLE;
- the video attributes: REVERSE and UNDERLINE.
- the colour attributes: BLACK, WHITE, BLUE, CYAN, GREEN, MAGENTA, RED and YELLOW.

Here is the detailed description of the colour, intensity and video attributes supported by the Querix 4GL statements:

Type	Attribute	Display Result	Example
Colour	WHITE	White colour (default font colour in character mode)	white
	YELLOW	Yellow colour	yellow
	RED	Red colour	red
	MAGENTA	Magenta colour	magenta
	BLUE	Dark-blue colour	blue
	GREEN	Green colour	green
	CYAN	Cyan colour	cyan
Intensity	BLACK	Black colour (default colour in GUI mode)	black
	NORMAL	Default colour (white for character mode and black for GUI mode) with no intensity applied	normal
	DIM	The colour of the display value becomes pale	dim
Video	BOLD	The colour is intensified in character mode and the font becomes bold in GUI mode	bold
	INVISBLE	This attribute prevents the corresponding 4GL statement from displaying information on the screen	N/A
Video	UNDERLINE	This attribute underlines the displayed values.	underline



REVERSE	This attribute displays the values in reversed video. The current colour of the font becomes the colour of the background and the colour of the background becomes the colour of the font.	reverse
---------	--	----------------

All of the attributes mentioned above except the attributes of the video attributes are mutually exclusive, which means that you can use only one attribute for the colour definition and one attribute for the font intensity. If you include two or more attributes specifying colour into one ATTRIBUTE clause, they will be in conflict. In such situation the position of the conflicting attributes determines which one will be used for information displaying: the last attribute usually benefits.

Only the REVERSE and UNDERLINE attributes which belong to video type can be present simultaneously within one ATTRIBUTE clause.

If you specify the INVISIBLE attribute to the INPUT or INPUT ARRAY statement, for example, the user will not see the information he is entering, but the information itself will nevertheless be stored in the input buffer.

Usage

The following example shows the usage of the attributes GREEN, UNDERLINE and REVERSE in the MESSAGE statement with ATTRIBUTE clause:

```
MAIN
  MESSAGE "We have used the attributes: GREEN, UNDERLINE and REVERSE"
  ATTRIBUTE (GREEN, UNDERLINE, REVERSE)
  CALL fgl_getkey()
END MAIN
```

Control Attributes

The control attributes are usually required when you need to define the behaviour of the application while executing the statement to which these attributes are assigned. Some of them, STYLE or IMAGE , for example, may also define the presentation of information displayed.

The detailed description of the control attributes mentioned in the table at the beginning of this chapter can be found within the sections dedicated to the statements that support them.

One should pay special attention to the attributes supported by INPUT, INPUT ARRAY, CONSTRUCT and DISPLAY ARRAY sub-dialogs, as they differ from the attributes that can be assigned to the same but standalone statements. The list of them for the DIALOG clauses is reduced since some functions are transferred to the DIALOG statement. More information about the peculiarities of the sub-dialogs' supported attributes is placed within the DIALOG statement chapter of this guide.

The most frequently used with the 4GL statements attributes are described below.

ACCEPT and CANCEL attributes

The ACCEPT and CANCEL attributes are used to indicate whether the default accept or cancel action should be added to the dialog window. The values of these attributes are of the Boolean data type and may be assigned to them in the following ways:

- ATTRIBUTE (ACCEPT/CANCEL = 1) or ATTRIBUTE (ACCEPT/CANCEL = 0),
- ATTRIBUTE (ACCEPT/CANCEL = TRUE) or ATTRIBUTE (ACCEPT/CANCEL = FALSE),
- using some program variables: ATTRIBUTE (ACCEPT/CANCEL = *var*)



The omitting of this attribute as well as the TRUE (1) value assigned to it means that the action will be added, the FALSE value will result in the action not registered.

HELP attribute

The HELP attribute is specified within the attribute clause to define the help message associated with the statement that is a literal integer or a program variable. For example:

- ATTRIBUTE (HELP = 78) : the help keystroke will result in displaying the help message number 78,
- ATTRIBUTE (HELP = *var*) : the help keystroke will result in displaying the help message number assigned to the program variable *var*.

NAME attribute

The NAME attribute is used to set the unique name to the statement within which it is assigned to make easier the further manipulations with it.

STYLE Attribute

The STYLE attribute is used to link a window, a form or a 4GL statement to a class filter specified in the current theme file. This allows changing the display properties according to the context without adding extra 4GL code.

The attribute needs the class name to be passed as a character value:

```
ATTRIBUTE (STYLE="my_class")
```

There exists a special "full-screen" class, that can be passed to an application as a style attribute value to enable a full-screen mode. Its usage is required when an application is expected to be run on different devices to keep the native look of its windows:

```
OPEN WINDOW w_full screen WITH FORM "window_full_screen" ATTRIBUTE (BORDER,  
STYLE="full-screen")  
  
# opens the window titled "w_full_screen" in a full-screen mode
```

If you are looking for the complete list of the classes with special meanings, please, refer to [Appendix B](#), further in this document.

Attribute Precedence

You can assign several attributes to the same form field using different methods. The precedence of execution will be as follows:

1. The ATTRIBUTE clause of the statement
2. The attributes from the ATTRIBUTES section of the corresponding form specification file.
3. The default attributes (they are specified in the sys collate table)
4. The latest OPTIONS ATTRIBUTE clause
5. The ATTRIBUTE section of the corresponding DISPLAY FORM statement
6. The ATTRIBUTE of the corresponding window specified in the OPEN WINDOW statement
7. Default colour settings of the terminal and the default line positions.

If a STYLE attribute is specified on any level, the priority is given to the theme attributes. The other display attributes specified on the same level will be ignored. More specific attributes specified on lower levels will be applied.

The statements that may refer to form fields are: DISPLAY, DISPLAY FORM, DISPLAY ARRAY, INPUT, INPUT ARRAY and CONSTRUCT, which are affected by the precedence rules listed above.



The ATTRIBUTE clauses of the statements ERROR, MESSAGE and PROMPT are not affected by the precedence rules.

 A circular icon containing a lowercase letter 'i' inside a purple circle, with a light purple shadow effect.	<p>ATTRIBUTE clause will produce an effect only if the termcap or terminfo file supports the attributes specified in the clause.</p> <p>The terminfo file is used on UNIX systems only and it does not support colour attributes. The only valid attributes for this file are REVERSE and UNDERLINE.</p> <p>Some terminal entries in these files include the sg#1 or xmc#1 capabilities. If you use such terminal and the attributes of a form or a window differ from the INPUT ARRAY attributes, the square brackets that are used as the delimiters of a field will be replaced with blank characters. The blank characters are used as transitional characters between attributes of different kind.</p>
--	--



Logical Blocks

It is now possible to define logical sub-blocks in 4GL code using the BEGIN .. END syntax:



Element	Description
Statement List	One or more executable statements, either 4GL or SQL
Declaration Clause	It is an optional DEFINE statement with the list of the declared variables

A logical sub-block may contain its own variable declarations, whose scope is restricted to the sub-block. Such sub-blocks can occur in the MAIN or FUNCTION program blocks as well as within complex statements which have clauses with executable statements: CASE, IF, FOR, FOREACH, INPUT, INPUT ARRAY, DISPLAY ARRAY, etc.

For example, we can insert a sub-block in the INPUT statement:

```
INPUT BY NAME customer_rec.*  
      AFTER FIELD fname  
      BEGIN  
          DEFINE i INTEGER  
          DEFINE val LIKE customer.fname  
          #variables 'i' and 'val' exist only within the scope of the current  
          #BEGIN...END block  
          LET val = get_fldbuf(fname)  
          SELECT COUNT(*) INTO i FROM customer WHERE customer.fname = val  
          IF i <> 0 THEN  
              ERROR "Must enter a unique name"  
          END IF  
      END  
      ...  
  END INPUT
```

Forms opened with the help of the OPEN FORM statement within such sub-block cannot be displayed with the help of the DISPLAY FORM statement outside the sub-block, as their scope of reference is the sub-block only. This is true for other objects such as cursors, reports, etc. To be able to use objects outside the BEGIN...END block, you should declare these objects with the GLOBAL scoping. Objects declared as global acquire global scope of reference which includes all the program modules. Objects open without scoping specification or with the LOCAL scoping cannot be references outside the logical sub-block. For more information about optional object scoping, see the "[Scoping of Objects](#)" section below.





Chapter 3: Statements

CHAPTER 3: STATEMENTS

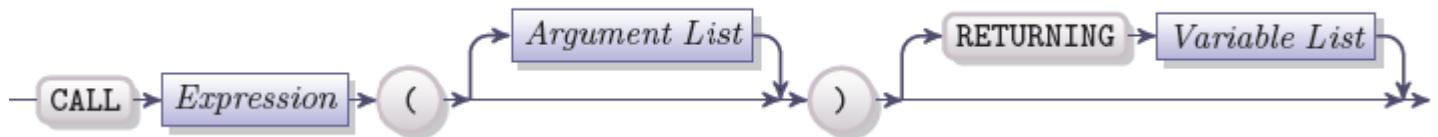
THIS CHAPTER WILL COVER ALL THE STATEMENTS WHICH CAN BE USED WITHIN A 4GL PROGRAM. IT WILL NOT GIVE MUCH DETAILS ABOUT THE STATEMENTS USED FOR REPORTS, AS THEY WILL BE DISCUSSED IN THE FOLLOWING CHAPTER. THE REST OF THE STATEMENTS WILL BE DISCUSSED IN DETAILS WITH ALL POSSIBLE RESTRICTIONS AND USAGE POLICIES.



CALL

The CALL statement can be used to invoke a function within a 4GL application. It can be 4GL functions, C functions, or ESQL/C functions. The 4GL functions can be built-in functions or those defined by the programmer.

Programmer-defined 4GL functions can be included within the primary 4GL application file, which contains the MAIN program block, or they can be contained within other .4gl source files and function libraries.



Element	Description
Expression	The name of the function you want to call
Argument List	The optional list of arguments which you want to pass to the called function, it can be empty
Variable List	It can be any variable of a simple or a structured data type (an element of an array or a record). It cannot be of a large data type.

Below is an example of calling a programmer-defined function:

```
MAIN
...
CALL my_function()
...
END MAIN
FUNCTION my_function()
...
END FUNCTION
```

When 4GL encounters a CALL statement that invokes a programmer defined function, it looks for the FUNCTION program block which corresponds to the function name specified in the CALL statement. A programmer defined function must have a unique name, otherwise 4GL will not be able to define which function to invoke and an error will occur. If a CALL statement contains the RETURNING clause, the invoked function can return values from the function into the calling routine.

The 4GL functions are not case sensitive and can be specified either in lower case or in upper case letters. The C functions are case sensitive; they must be specified in lower case letters



The Argument List

When the CALL statement is used, it must be followed by a function name. The function name must always be followed by a list of actual arguments for the statement to pass on to the function. The list of arguments must be enclosed in parentheses. If no arguments are specified, an empty Argument List placed between parentheses must still be included:

```
CALL function_name()
```

The actual arguments in the Argument List will be passed to the formal arguments of a function by value. The actual arguments specified in the CALL statement must match the formal arguments specified next to the function name at the beginning of the FUNCTION program block in number and order, they also must be of the same or compatible data types.

```
MAIN
...
CALL my_function (my_rec.fname, my_rec.lname)
...
END MAIN
FUNCTION my_function (p_fname, p_lname)
DEFINE p_fname, l_fname VARCHAR(15)
...
END FUNCTION
```

Variables of simple data types, elements of program arrays, members of program records, as well as large data types can be used as actual arguments in the Argument List of the CALL statement.



Note: If you use a variable that does not have a value yet as an actual argument in a CALL statement, you may get unexpected results.

Passing Arguments

The arguments are usually passed to the function by value, but they may also be passed by reference. It depends on the data type of the argument.

The following arguments are passed by value:

- arguments of simple data types
- arguments of structured data types (array elements and record members)

If a value of an argument passed by value is changed within a function, this change does not affect the value of this variable used in the calling routine, unless this value is returned by means of the RETURNING clause.

The arguments of large data types (TEXT and BYTE) cannot be referenced by value, so they are passed to the function by reference. The function performs changes using the actual values of TEXT or BYTE variables, which are also used by the calling routine. Thus the changes made in the FUNCTION program block affect the value of the corresponding TEXT or BYTE variables in the calling routine. The RETURNING clause cannot be used to return the values of the large data types.



The RETURNING Clause

The RETURNING clause assigns the values returned by the function that has been called to variables in the calling routine. The RETURNING clause is optional; a function may not return any value. The variables in the Variable list of the RETURNING clause should be separated by commas. E.g.:

```
RETURNING arr1[54].elem5, my_rec.* , var32
```

The RETURNING statement can assign values to the receiving variables only if there is the RETURN clause in the called function. Values listed in the RETURN clause should correspond in number and position to the variables listed in the RETURNING clause, they also should have compatible data types. If there are more variables in the RETURNING clause of the CALL statement than the RETURN clause contains, an error will occur. If the RETURNING clause contains fewer variables than are contained in the RETURN clause, all the excessive variables of the RETURN statement are ignored.

The RETURNING clause receives information from function by value. Because variables of the BYTE and TEXT data types can only be passed by reference, they cannot be included in the RETURNING clause.

```
MAIN
...
CALL subtraction (a,b) RETURNING c
...
END MAIN
FUNCTION subtraction (p_a, p_b)
DEFINE p_a, p_b, r_c INT
LET r_c = a-b
RETURN r_c
END FUNCTION
```



Note: If a function doesn't return anything, the RETURNING clause should be omitted. It can also be omitted, if the CALL statement invokes a function that returns values, but these values are not referenced by any statement in the calling routine.

The Restrictions on Returned Character Strings

A returned value of character data type cannot be longer than 511 bytes. No more than 10 lines 511 bytes each can be returned by a function. The actual size of the returned character strings is limited to 512 bytes, but they require the terminating ASCII 0 at the end.

To pass longer values by reference, use the variables of TEXT data type which are passed to the calling routine without the RETURNING clause.



Invoking a Function without the CALL Statement

It is possible to invoke a function that returns exactly one value without using the CALL statement. You can simply include the function (and the list of its formal arguments, if they are needed) into any expression, where the returned value will be valid. In the example below, a function that returns one value is used in the IF statement:

```
IF subtraction(a,b) > 0 THEN
    DISPLAY "Passed"
ELSE
    EXIT PROGRAM
END IF
```

The Coma and Double-Pipe Symbols

Two symbols can be used in the list of actual arguments of the CALL statement: coma (,), and double-pipe (||) symbols. These symbols can be used in function calls both made by means of the CALL statement and without it.

If the CALL statement passes more than one actual argument to a function, the arguments are separated by commas. Unlike in the LET statement, the coma symbol does not have the concatenation semantics in this context, it only serves as a separator sign.

You can use a double-pipe symbol, which is a concatenation operator, to concatenate two values. In this case, the two values will be passed as one value to a single formal argument of a function.

```
CALL my_function (var1 || var3, var2)
```

The above example passes two values to the function: the value of *var2* and the concatenated value of *var1var3*.



CASE

The CASE statement allows a range of conditions to be applied to data selection. It is possible to include more than two conditions into the CASE statement, which makes it more powerful than the IF statement. CASE statement is similar to a set of nested IF loops.



Element	Description
Expression	It is an optional 4GL expression
Declaration Clause	An optional clause where values and functions local to the CASE statement can be declared
WHEN Clause	One or more WHEN clauses which contain a 4GL expression and a number of executable statements

There are two types of CASE statements depending on whether the optional 4GL expression follows the CASE keyword:

Type	Usage
I type: CASE statement is followed by a 4GL expression	You must specify an INT, SMALLINT, DECIMAL, CHAR(1) or VARCHAR(1) expression in the WHEN block. 4GL executes the WHEN clause, if the expression following CASE statement and the expression specified in this WHEN clause return the same value and if that value is not NULL.
II type: CASE statement is not followed by an expression	Boolean expressions must be specified in the WHEN clauses. If it is evaluated as TRUE, the WHEN statement is executed.

Here is an example of the CASE statement type I:

```

CASE result
    WHEN 1
        CALL open()
    WHEN 2
        EXIT CASE
    OTHERWISE
        CALL retry()
    END CASE
  
```

The next example is an example of the CASE statement type II:

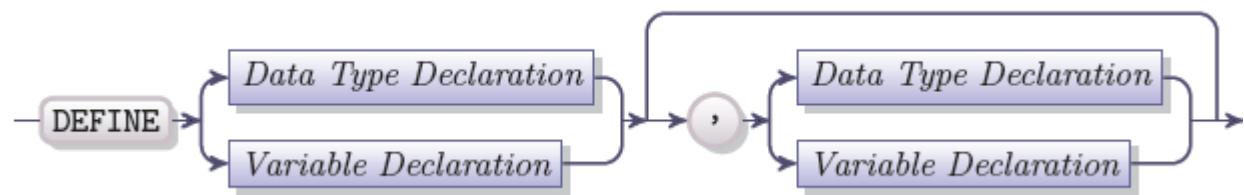


```
CASE
    WHEN answer MATCHES "[Yy]"
        CALL open()
    WHEN answer MATCHES "[Nn]"
        EXIT CASE
    OTHERWISE
        CALL retry()
    END CASE
```

The END CASE statement must indicate the end of the CASE statement, it must be placed after all WHEN blocks and after OTHERWISE block, if it is present.

The Declaration Clause

The Declaration clause of the CASE statement has the following structure:



Element	Description
Data Type Declaration	A block where a user-defined data type is declared (for more information see the " Data Type Declaration " section of the DEFINE statement)
Variable Declaration	A block where the data type of a variable is declared (for more information see the " Variable Declaration " section of the DEFINE statement)

The scope of reference of the variables or function prototypes declared in the Declaration clause of the CASE statement is this CASE statement. They cannot be referenced outside the CASE statement in which they are declared.

The example below represents a CASE statement with the DEFINE clause:

```
CASE a
    DEFINE b INT
    PROTOTYPE FUNCTION my_function (b)
    WHEN 1
    ...
    WHEN 2
    ...
    OTHERWISE
```

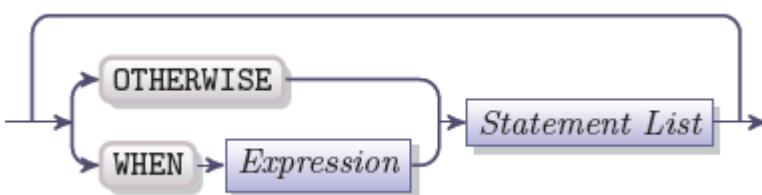


```
...
END CASE
```

The variable "b" and the function "my_function" can be referenced in the WHEN clauses of the CASE statement.

The WHEN Clauses

A WHEN clause must specify an expression, it may be an expression that returns either an INT, SMALLINT, DECIMAL, CHAR(1) or VARCHAR(1) value for the type I of the CASE statement or a Boolean expression for the type II of the CASE statement. A WHEN clause must also contain one or more associated executable statements.



Element	Description
Expression	<ul style="list-style-type: none"> For the CASE type I – INT, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1) expression For the CASE type II – a Boolean expression
Statement List	The list of executable statements

The type of data that can be returned is specified by what comes after the CASE keyword (by the type of the CASE statement):.

- If the CASE keyword is followed by only one expression as a selection criterion (type I), the WHEN must be followed by an INTEGER, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1). If a WHEN expression matches the value of the CASE expression, 4GL executes the statements of this WHEN block and then exits the CASE statement. If CASE expression returns NULL or FALSE, 4GL does not execute any of the WHEN clauses.
- If the CASE is not followed by an expression (type II), the expressions in the WHEN clauses are treated by 4GL as Boolean expressions. If a Boolean expression of one of the WHEN clauses returns TRUE (that is, not zero or NULL), 4GL executes the statements in this WHEN clause. If the Boolean expression of a WHEN clause returns FALSE or NULL, 4GL does not execute such WHEN clause.

If a condition applies to more than one WHEN clause, 4GL executes the first WHEN block in the list that conforms to the condition and then exits the CASE statement.

There is an implied EXIT CASE statement following the statements of each WHEN clause. Program control will pass to the first statement following the END CASE keywords after it has executed the all the statements of the corresponding WHEN clause.



The OTHERWISE Clause

The OTHERWISE block is executed when the condition of neither of the WHEN clauses is met. This means that none of the expressions of the WHEN clauses is evaluated as TRUE, or none of them match the value of the CASE *expression*.

The OTHERWISE clause is optional, if it is included into the CASE statement, it should be placed after the last WHEN clause. Each CASE statement can have only one OTHERWISE clause.

If the OTHERWISE clause is used, it implicitly contains an EXIT CASE statement no matter whether it is included in the clause or not, and it passes control to the first statement that follows the END CASE keywords after 4GL has executed all the statements in the OTHERWISE clause. The only case when the OTHERWISE clause does not pass control to the line following the END CASE statement is when the clause contains a valid GOTO statement.



Note: The usage of the GOTO statement in a WHEN clause can sometimes cause the -4518 runtime error. It is advisable that the EXIT CASE statement (implicit or explicit) be used to leave a WHEN clause.

The END CASE and EXIT CASE Keywords

The EXIT CASE statement terminates the execution of a WHEN clause or an OTHERWISE clause and passes program control to the first statement following END CASE keywords. 4GL skips any statements located between EXIT CASE and END CASE keywords.

END CASE keywords define the end of the CASE statement; they must be placed after the last WHEN clause or after an OTHERWISE clause, if it is included.

For more details on the END CASE and EXIT CASE keywords, see the sections about the END statement and EXIT statement.

Using Boolean Expressions with the CASE Statement

Boolean expressions cannot be used in WHEN clauses of the CASE statement (type I). The following usage of Boolean expressions is **incorrect** but it could not be spotted while compiling:

```
CASE result
    WHEN result=A --incorrect
    ...
    WHEN result=B --incorrect
    ...
END CASE
```

The expression of the WHEN blocks of the CASE statement (type I) should return INT, SMALLINT, DECIMAL, CHAR(1), or VARCHAR(1), which does not happen, if a WHEN block contains a Boolean expression. Such structures can cause unexpected runtime errors, while 4GL compares CASE *expression* to the whole Boolean expression and not to the value returned by the Boolean expression. E.g. in the example above the value returned by the *result* variable is "A", but the first WHEN clause is not evaluated as TRUE, because *result* value is compared to "result=A" expression and not to "A".



CLEAR

The CLEAR statement is used to clear any of the following display sections:

- 4GL screen (**not** including 4GL windows)
- A specified 4GL window
- All of the fields of the currently active form
- One or more specified fields in the currently active form

	Note: CLEAR statement does not change or delete any information in the program, the values cleared from the screen will not be deleted.
--	--

The CLEAR FORM Statement

CLEAR FORM statement clears all the fields of the current form but does not affect the other parts of display and other sections of the window in which the form has been opened.

► **CLEAR FORM** ➔

To clear one or several fields of the form use CLEAR <field-list> statement.

CLEAR FORM

CLEAR WINDOW *window-name*

To clear a specified 4GL window, which does not need to be the current window, use the CLEAR WINDOW statement:

► **CLEAR WINDOW** ➔ **Window Name** ➔

Element	Description
Window Name	A name of the window you want to clear. It must be the same name which have been used in the OPEN WINDOW statement

If the window specified with the CLEAR WINDOW statement has a border, the border will still be displayed. CLEAR WINDOW statement does not depend on which window is currently active.

The CLEAR SCREEN Statement

CLEAR SCREEN option is used to make the 4GL screen the current window (that is to move it to the top of the window stack), and after that to clear it.



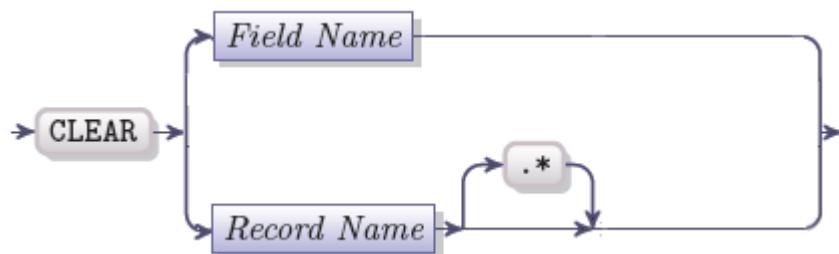
—CLEAR SCREEN →

The PROMPT, MESSAGE and ERROR lines will also be cleared.

CLEAR SCREEN

Clearing a Field

To clear a specified field or fields of a screen form, use the CLEAR keyword followed by the field list.



Element	Description
Field Name	One or more fields of the current screen form separated by commas.
Record Name	The name of a screen record
Character Expression	A character expression returns one or more (separated by comma) field names

The field names should be separated by commas, if the field list contains of more than one field. The example below clears two fields of a screen form: *name* and *country*. The values of any other field of the same form will remain in their places.

CLEAR name, country

To clear all the fields of a screen record, use the *screen_record.** notation. The example below clears all the fields that are linked to the *order_info* table within the currently active form:

CLEAR order_info.*

If you want to clear all the fields associated with a database table, use the *table.** notation instead of the *record.**.

The next example clears all the fields that are not associated with a table column and thus belong to the default screen record *formonly*.

CLEAR formonly.*



CLOSE DATABASE

The CLOSE DATABASE statement is used to close the current database, which has been opened with the DATABASE statement.

The CLOSE DATABASE statement is used without the identification of the database, because it closes the currently used database. After the database has been closed no operations can be performed with tables, though other statements not connected with the database can be used normally. If the database has been closed, the only valid SQL statements are: CREATE DATABASE, DATABASE, DROP DATABASE, ROLLFORWARD, DATABASE, and START DATABASE.

CLOSE DATABASE

Use CLOSE DATABASE statement before you DROP the database. PREPARE statement cannot contain CLOSE DATABASE statement.



CLOSE FILE

This statement closed the file opened earlier by the OPEN FILE statement. It frees the file descriptor variable and it can then be used in another OPEN FILE statement. The WRITE, READ and SEEK statements cannot reference a file descriptor that was used in a CLOSE FILE statement unless another OPEN FILE statement initiates it again.

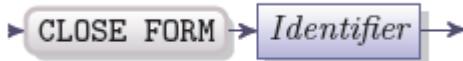


Element	Description
Descriptor	The integer variable that was previously used in an OPEN FILE statement.



CLOSE FORM

The CLOSE FORM statement must be used to regain access to the memory allocated to a form when 4GL opens it in by means of an OPEN FORM statement.



Element	Description
Identifier	The name of the form which has been used for this form file in the previously used OPEN FORM statement

If a form is opened with the help of OPEN FORM statement, the CLOSE FORM statement with the same form name must be used to close the form.

If a form has been opened with the help of the OPEN WINDOW statement with the WITH FORM clause, the form will be closed on CLOSE WINDOW statement together with the window and no CLOSE FORM statement is required.

```
OPEN WINDOW window1 AT 2,2 WITH FORM "form1"  
...  
OPEN WINDOW window2 AT 15,15 WITH 10 ROWS, 35 COLUMNS  
OPEN FORM form2 FROM "form2"  
DISPLAY FORM form2  
...  
CLOSE FORM form2  
CLOSE WINDOW window2  
CLOSE WINDOW window1
```

In the example above the form *form1* is opened by means of the OPEN WINDOW...WITH FORM statement and thus it is closed by the CLOSE WINDOW *window1* statement. The form *form2* is opened with the OPEN FORM statement and then displayed by means of the DISPLAY FORM statement. Thus to clause this form the CLOSE FORM statement is used.

If the form has been closed and you try to display it again with the help of DISPLAY FORM statement, an error message will appear. The form is no longer stored in the memory and to display it once more you will need to use OPEN FORM statement again.

If OPEN FORM or OPEN WINDOW...WITH FORM statements are used for the form that is already open, the form will be closed and then reopened.



CLOSE WINDOW

CLOSE WINDOW statement closes a specified 4GL window; it does not have to be the current window.



Element	Description
Window Name	A name of the window you want to close. It must be the same name which has been used in the OPEN WINDOW statement

A valid Window Name provides the CLOSE WINDOW statement with a static window identifier which cannot be changed throughout the program

CLOSE WINDOW statement clears the specified window from the display and restores the part of the screen which has been covered by this window. It also frees the memory allocated to the window by OPEN WINDOW statement. If the window has been opened using WITH FORM section of the OPEN WINDOW statement, CLOSE WINDOW statement closes both the window and the form.

Closing a window has no effect on any variables that were set while it was open.

If the window is currently being used for input, using CLOSE WINDOW will result in a runtime error. This means that CLOSE WINDOW cannot be used while any of the following statements are being executed: CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, or MENU.

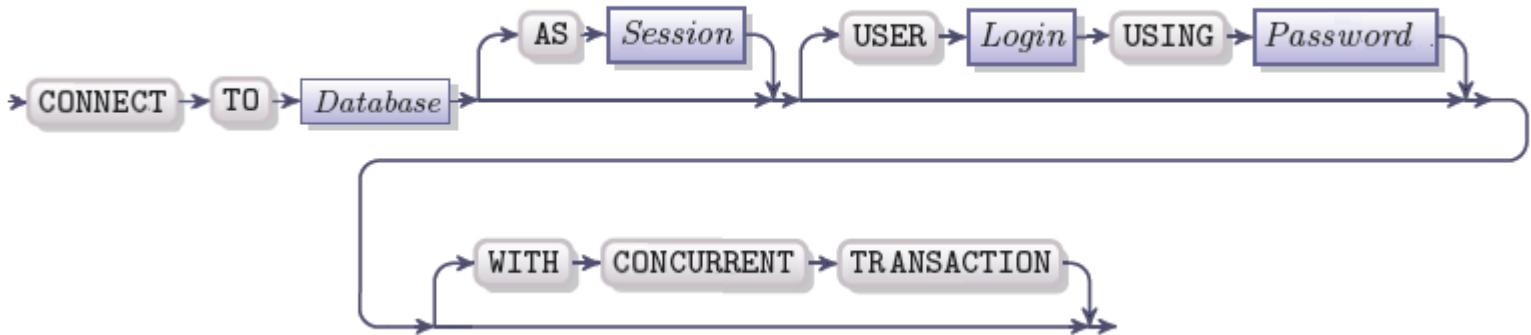
CLOSE WINDOW dialog_box

When a window is closed, the corresponding changes are made in the window stack. A window stack is the order in which the opened windows are perceived by the 4GL. When a new window is opened it is added to the top of the window stack and becomes the current window. When the current window is closed by means of the CLOSE WINDOW statement, it is deleted from the stack and the window below it is made the current window and moved to the top of the stack. If the CLOSE WINDOW closes other but the current window of the window stack, this window is deleted from the stack and the windows in the stack are shifted so that the empty place is taken up by the next window. Closing of a non-current window has no effect on the current window display.



CONNECT TO

The CONNECT TO statement allows a program to connect to several databases at a time. The databases can be located on different servers and be of different types. The syntax of this statement is as follows:



Element	Description
Database	A string expression indentifying the name of the database to connect to
Session	A string expression indentifying the session name for the connection
Login	A string expression indentifying the username valid on the database server
Password	A string expression indentifying the password valid for the username used

The CONNECT TO statement establishes the connection to a database. A program can have several CONNECT TO statements located within program modules. Unlike the DATABASE statement, this statement cannot appear before the MAIN block. Use the [DATABASE](#) statement to specify the compile time database, if you need to compile DEFINE ... LIKE statements. The CONNECT TO statement is only used to specify the runtime database connections.

The current database connection that will be active is the session of the last CONNECT TO statement executed. The current connection can be changed with the help of the [SET CONNECTION](#) statement and closed using the [DISCONNECT](#) statement.

If you connect to Informix database running on a UNIX platform, the only restriction on establishing multiple connections to the same database environment is that an application can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared-memory connection mechanism or the local-loopback connection mechanism, examine the \$INFORMIXDIR/etc/sqlhosts file.

Database Name

The database name will be typically the name of the database on the database server. The name should be included into the quotation marks, if the literal string is used.



Connection Preset in Database Configuration File

If the database connection you want to use is set in the database.cfg file (for more information about the file see the "Lycia II Developers Guide") , the database name used in the CONNECT TO statement should be the same as the database alias used in the configuration file.

Below is an example of the database.cfg file content:

```
ifx_db {  
    driver = "informix"  
    source = "stores@ixserv"  
    username = "jane"  
    password = "567890"  
}
```

To connect to the database specified in the configuration file under the alias 'ifx_db', the following CONNECT TO statement should be used:

```
CONNECT TO "ifx_db"
```

It is equivalent to the following DATABASE statement, though the DATABASE statement will limit the database connections to this database only, while the CONNECT TO statement allows you to specify other connections:

```
DATABASE ifx_db
```

Upon executing either of these statements Lycia will check the database.cfg file for the database alias 'ifx_db'. If the alias is found, the connection details associated with it such as the driver and the actual database name and server as well as the login and password will be used during the connection. All the parameters of the alias in this file are optional. If some are missing, the values of the corresponding environment variables will be taken in consideration as well. For more details on handling the optional parameters this see the Lycia II Developer Guide.

The database alias used in the CONNECT TO statement need not be the same as the actual name of the database, if the alias is listed in the database.cfg.

Connection Absent from Database Configuration File

If the database name used is not found in the list of the aliases in the database configuration file, Lycia will presume that the database name used with the CONNECT TO statement is the actual database name and will try to look for this database specified by the default database driver which was set by the LYCIA_DB_DRIVER variable.

Specifying Additional Parameters in the Database Name

It is possible to override both the default driver and the driver preset for a database in the database configuration file. In this case the database name can be followed by the plus sign (+) and then the 'driver=<driver name>' string. Here is the syntax of the database name in this case:

```
CONNECT TO "db_alias+driver='informix'"
```

In this case, if the db_alias was specified in database.cfg with driver="oracle", the above code will try to open the database with the parameters specified in the file not on the Oracle server but on Informix server instead.



It is also possible to put all the required parameters which are normally stored in the database configuration file into the database name string. It is not advisable to do so other than for the testing purposes, since hardcoding the username and password is not secure. For example:

```
CONNECT TO "db_alias+driver='informix' username='user', password='12345'"
```

Connecting to Default Database

If the keyword DEFAULT is used instead of the database name, Lycia will connect to the default database server defined by the INFORMIXSERVER environment variable. This option is available only for Informix databases.

```
CONNECT TO DEFAULT
```

Connection Session

The optional AS clause allows you to give the connection session a specific name. If this clause is omitted, the session name is the same as the database name used. This clause is used to open two and more connections to the same database.

It is not allowed to create two sessions with the same name - the session name must be unique. If this clause is omitted, two connections to the same database will have the same session name and will cause an error. Use the AS clause to change the session name, e.g.:

```
CONNECT TO "my_db" -- session name "my_db"  
CONNECT TO "my_db" AS "session1" -- session name "session1"  
CONNECT TO "my_db" AS "session2" -- session name "session2"
```

	Note: The session name is case sensitive.
---	--

Login and Password

The optional clause USER ... USING allows you to set the username and password for a database user at runtime. This makes the database connection more secure than specifying them in the database configuration file. If the username and password were already set in the database.cfg file for the specified database name, using this clause in the CONNECT TO statement will override the preset credentials.

This way it is possible to set the default username and password that will be used during compilation, in case you need to compile some DEFINE...LIKE statements, and then at runtime switch to another user. Thus the default user may have minimal permissions which will not compromise the security even though the username and password will be stored in configuration file.

Here is an example of using the USER... USING clause:

```
CONNECT TO "my_db" USER "john" USING "123456"
```

If the credentials are not set either in the CONNECT TO statement or in the database configuration file, the program will attempt to establish connection without the username and password and will most probably fail. You can also connect to the same database with different users at the same time:

```
CONNECT TO "my_db" AS "usr1" USER "john" USING "123456"
```



```
CONNECT TO "my_db" AS "usr2" USER "jane" USING "567890"
```

The WITH CONCURRENT TRANSACTION Keywords

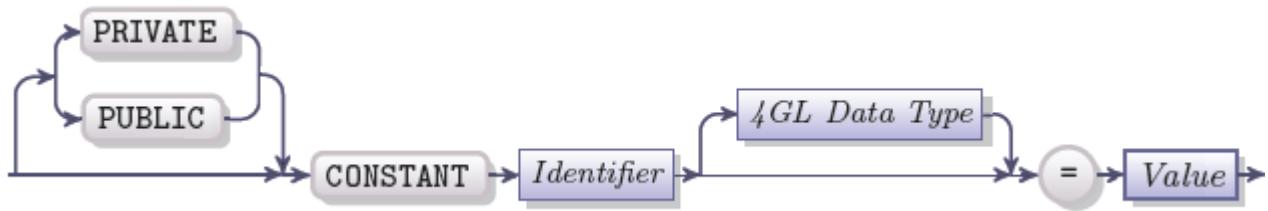
If these keywords are present, they allow a program to open several transactions concurrently in different database sessions. So several transaction sessions can be open at the same time, if the connection to the databases involved in them was opened using these keywords. Otherwise only one transaction at a time is allowed.



CONSTANT

A CONSTANT statement defines a constant which, like a variable has a name, but unlike a variable also has a predefined and unchangeable value. Constants can be useful for declaring special symbols and symbol combinations such as "\n".

The CONSTANT statement has the following syntax:



Element	Description
Identifier	The name of the constant. It should be unique and differ from the names of the variables available in the same scope of reference.
Datatype	The optional data type specification of the constant
Value	The unchangeable value of the constant

Just like variables, constants can have global, module and local scope of reference depending on the position where they are declared. The constants can be used throughout the 4GL code within their scope of reference after they were declared. Constants can be passed to functions and returned by them.

The data type definition for a constant is optional. If it is not specified, the literal value assigned to the constant defines its data type implicitly. For example:

```
CONSTANT const1 = 5 -- the data type is defined automatically as INT
CONSTANT const2 = "yes" -- the data type is STRING
CONSTANT const3 CHAR(3) = "5" -- the data type is CHAR(3)
```

	Note: If you explicitly specify one of the character data types for a constant, make sure that the constant value is enclosed in the quotation marks. Since the implicit data type conversion does not occur with the constants, assigning a number to a constant without enclosing it in quotes will cause an error, whereas assigning it to a character variable will just result in automatic conversion.
--	---

If the data type is declared and it has the scale and/or precision, make sure that the literal value assigned corresponds to the precision. For example, if you assign value "abcde" to a constant with the data type declared as CHAR(3), the content of the constant will be truncated.

The Scope of Usage

Constants can be used in the consequent DEFINE statements, e.g.:



```
CONSTANT i = 10
DEFINE ar ARRAY[i] OF INT
```

With other 4GL statements constants are usually utilized in places where the literals would normally be placed:

```
CONSTANT n = 100
...
FOR i = 1 TO n
```

Naturally, if a constant was not defined using the CONSTANT statement and then used in the code, a compile-time error will be thrown.

Private and Public Constants

The constants defined at the beginning of a module have module scope of reference by default. They are visible and can be used only in the current module. You can declare a private constant explicitly by specifying the PRIVATE keyword before the CONSTANT statement.

To declare a public constant, specify the PUBLIC keyword before the CONSTANT statement. A public constant can be referenced by other modules of the same program. It can also be referenced, if the module is imported using the [IMPORT](#) statement.

Usage Restrictions

There are several restrictions as to in which statements and situations constants cannot be used:

- It cannot be used in the ORDER BY clause of the SELECT statement.
- It cannot be used in cases where the automatic conversion of data types takes place - constants are not converted automatically. Thus if you have a constant of a character data type with value "123", you cannot use it in an arithmetic expression:

```
CONSTANT a CHAR(3) = 123
DISPLAY 100+a -- you will get 100 as a result
```

For the same reason the constant above cannot be used in other cases where an integer value is expected, e.g. as the size of an array during the array declaration, as a counter index in the FOR statement and so on.

- Values cannot be assigned to a constant in any way. The following line will throw a compile time error:

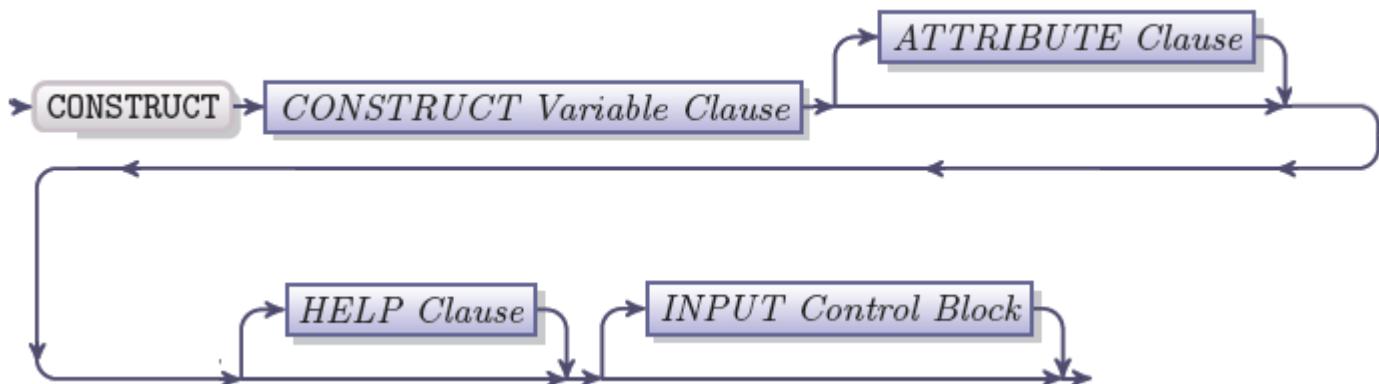
```
CONSTANT a = 123
LET a = 567 -- this line will cause an error
```





CONSTRUCT

The CONSTRUCT statement is created to run a query by example. The user can run a query on a database by specifying a value or a range of values for the screen fields that are associated with the columns in that database. The CHAR, STRING or VARCHAR variable that results from the CONSTRUCT statement contains the condition for the WHERE clause of the SELECT statement.



Element	Description
CONSTRUCT Variable Clause	The clause that contains a character variable used for storing the user entered search criteria, the corresponding form fields and column names
ATTRIBUTE Clause	An optional ATTRIBUTE clause where the attributes for the fields used in the CONSTRUCT statement can be specified
HELP Clause	An optional HELP clause where the help number for the CONSTRUCT statement can be specified
INPUT Control Block	An optional CONSTRUCT INPUT control block which controls the input of the data for the query

The CONSTRUCT statement permits the user to make a query by example by specifying ranges of values for fields of a screen form that correspond to table columns in a database. These values are converted to a single Boolean expression that may appear in WHERE clause of a prepared SELECT statement to specify search criteria. The CONSTRUCT statement can also control the environment in which search criteria are set with the help of the INPUT control block.

The following preparations should be made in order to use the CONSTRUCT statement:

Define fields in the form file and associate them with the database columns

Define a variable of CHAR, STRING or VARCHAR data type

Open and display the screen form with the help of either OPEN FORM and DISPLAY FORM statements or the OPEN WINDOW statement with the WITH FORM clause.

After that you can use a CONSTRUCT statement to store a Boolean expression based on the user-entered search criteria.

When the CONSTRUCT statement is used, the current form is activated. It is the most recently displayed form. If you use more than one 4GL window, it is the form displayed in the currently active window. The current window can be set with the help of the CURRENT WINDOW IS statement.



As 4GL encounters a CONSTRUCT statement at runtime, it performs the following actions:

- Clears all the fields of the form contained in the FROM clause of the CONSTRUCT statement, or implied by the BY NAME clause
- Executes statements in BEFORE CONSTRUCT block of the CONSTRUCT statement, if such block is present
- Moves screen cursor to the first field of the Field list

The user is supposed to enter some values into the fields of the screen form. If no value is entered, any value found in the corresponding column of the specified database will match search criteria.

After the user has entered the values and pressed the Accept key, the CONSTRUCT statement uses AND operators to combine the entered values into a single Boolean expression that is stored in the variable of CHAR, STRING or VARCHAR type. 4GL will assign the TRUE expression "1=1" to the character variable, if no search criteria are entered.

The variable resulting from the CONSTRUCT statement can be used in a WHERE clause, to search the database for matching rows, if the resulting variable is combined with an SQL statement which performs a query. Usually it is combined with the SELECT statement.

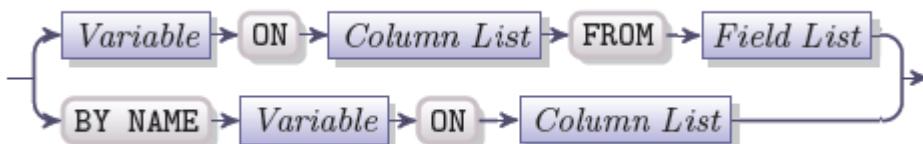
The SELECT statement must be prepared together with the resulting variable by means of the PREPARE statement. To execute the prepared statements you can:

- Use the EXECUTE statement for any SQL statement and SELECT statement **with** the INTO clause
- Use an SQL cursor (with the DECLARE and FOREACH statements or with the OPEN and FETCH statements) for a SELECT statement **without** the INTO clause

When 4GL encounters the END CONSTRUCT keywords, the form is cleared. The content of the Boolean expression is not affected by environment variables used to format data values, such as: DBDATE, DBTIME, DBFORMAT, DBFLTMASK, and DBMONEY.

The CONSTRUCT Variable Clause

The CONSTRUCT variable clause is used to specify the character variables that can store user-entered search criteria and the form fields in which the user should enter these criteria. The following diagram describes Variable clause and its sub-clauses:



Element	Description
Variable	A variable of a CHAR, STRING or VARCHAR data type that stores a Boolean expression used to summarize the user-entered search criteria.
Column List	The list of one or more columns in the database separated by commas
Filed List	The list of one or more form fields separated by commas which are



used for entering the search criteria

The Variable clause associates the specified fields with the specified columns in a database.

	In the variable clause of the CONSTRUCT statement, VARCHAR variables are treated as CHAR variables. It means, that if the value specified to a VARCHAR variable is shorter than the declared variable size, the extra space will not be truncated (as with the usual VARCHAR treatment), but the assigned value will get trailing spaces to fit the declared size.
---	--

The fields can be associated implicitly, with the help of BY NAME keywords, or explicitly, by identifying the field list with the help of the FROM clause. A field must be of data type that is compatible with the data type of the corresponding column.

The order in which the cursor moves from field to field is predefined by the order of the fields in Field list of the FROM clause. It is possible to specify only one screen record within one screen array.

A table reference in the field list within the FROM clause cannot include table qualifiers, defining a database, an owner of the table or a server. An alias must be declared in the form specification file for any table reference that requires a qualifying prefix (such as database, server, or owner). Alias is declared in the TABLES section of the form specification file.

A variable of character data type that results from a CONSTRUCT statement links each column name with the search criteria the user enters in the corresponding form fields specified in the FROM clause or implied by the BY NAME clause. To avoid overflow, the length of this variable should be several times the total length of all the fields. It should be done because besides the information from the fields a Boolean expression includes additional operators to combine the user-entered values into a single value.

The BY NAME Clause

The BY NAME clause can be used if the form fields have the same names as the corresponding columns in the database table (that is in the ON clause). The user can perform query only by means of the screen form fields implied by the BY NAME keywords. If at least one of the fields you want to use for the query does not match the column name, the FROM clause must be used instead of the BY NAME clause.

```
CONSTRUCT BY NAME search_1 ON name, city, country, phonecode ...
```

The CONSTRUCT statement takes place at the first row of a screen array, if the column names are associated with the field names that are included into a screen array. If it is to happen in any other row of the screen array, the FROM clause must be used. Column names cannot be preceded by table qualifiers (database server name, owner of the table, etc.) if BY NAME clause is used. If you need to specify table qualifiers, specify table aliases in the FROM clause.

The ON Clause

The ON clause is used to specify the names of the database columns for which users can enter search criteria. Columns must belong to the same database, but they may belong to different tables. Tables can be either in the current database specified by the DATABASE statement, or in the database specified in the DPPATH environmental variable

If a CONSTRUCT statement is to be used with the BY NAME keyword, the form fields must have the same names as the columns listed in the ON clause. If a CONSTRUCT statement is to be used with a FROM clause,



the list of column names in the ON clause must be equal in number and order to the fields listed in the FROM clause.

The expression *table.** can be used to specify every column in the given table, it can represent the whole column list or a part of it.

```
CONSTRUCT search_2 ON address.*, order_num, clients.* FROM...
```

The order of columns within table is dependent on their positions in the system catalogue table at the time the program is compiled. If you have changed the order of the columns with the help of the ALTER TABLE statement you might need to modify your CONSTRUCT statement.

The FROM Clause

The FROM clause contains the list of the field names or screen records which are linked to columns in a database table. It cannot be used, if the variable clause of the CONSTRUCT statement contains the BY NAME clause. It is required if:

The names of fields used in the CONSTRUCT statement differ from the column names to which they are linked

You need to specify additional table qualifiers in the ON clause

You want CONSTRUCT statement to take place in any row of a screen array beyond the first one

You want to change the default order of the form fields (the default order is determined by the order of the fields in the screen record)

The field list includes an alias that represents a table, view, or synonym name including any qualifier

The cursor can be placed only in the fields listed in the FROM clause (if the CONSTRUCT statement requires the FROM clause). The order and number of the fields in the FROM clause must correspond to the order and number of the names of the columns in the ON clause. If a screen record with an asterisks (*record.**) is used, make sure that the columns in the ON clause correspond in order and number to the record members.

To use screen array field names in the FROM clause, the row in which the CONSTRUCT is carried out has to be specified with the notation: *screen-record [line].field-name* where *line* is an integer greater than zero. If no *line* value is specified, it will default to the first screen record of the array.

```
LET i=1+step_var
CONSTRUCT search_2 ON client.* FROM scr_client[i].*
```

Here is an example of an alias declared in TABLES section in a form specification file ('my' is the qualifier of the owner of the table 'client').

```
CONSTRUCT search_3 ON my.client.fname, my.client.lname, my.client.company
FROM cl.fname, cl.lname, cl.company
```

Whereas the form specification file contains the following code:

```
TABLES
cl = my.client
```



An alias can be specified in the Graphical Form Editor of the LyciaIDE, if the form is in the .4fm format. For details see the "Lycia Developers Guide".

The ATTRIBUTE Clause

The display attributes for the fields, specified by the FROM clause or by the BY NAME clause are defined in the ATTRIBUTE clause of the CONSTRUCT statement. The ATTRIBUTE clause of the CONSTRUCT statement has the structure of the common [ATTRIBUTE clause](#). If the ATTRIBUTE clause is used, it overrides all the other attributes that are applied to the same fields:

- Default attributes used in the form specification file
- Attributes listed in the syscolatt table
- The NOENTRY and AUTONEXT attributes in the form specification file

If a field used in the CONSTRUCT statement has an AUTONEXT attribute, the CONSTRUCT statement ignores the AUTONEXT attribute without regard to whether the ATTRIBUTE clause is present or not, allowing the user to query for large ranges, alternatives, etc.

The ATTRIBUTE clause overrides any display attributes set by the INPUT ATTRIBUTE clause of an OPTIONS or an OPEN WINDOW statement for the duration of the CONSTRUCT statement execution. The attributes defined by the ATTRIBUTE clause of the CONSTRUCT statement are applied to all the fields of the form while it is active. After the form has been deactivated, its previous attributes come into effect.

```
CONSTRUCT search_2 ON client.* FROM scr_client[i].*
    ATTRIBUTE (REVERSE, GREEN)
```

An attribute specified in the ATTRIBUTE clause can produce the effect only if the termcap or terminfo files and the physical terminals support this attribute. On UNIX systems using terminfo files, colour specifying attributes are not supported and the only possible attributes in this case are: REVERSE and UNDERLINE.

The HELP Clause

The HELP clause is an optional clause that specifies the number of the help message that is associated with the CONSTRUCT statement.



Element	Description
Integer Expression	A 4GL expression that returns a positive integer greater than 0 and less than 32,767.

The message may contain the instructions for the user on how to make a query. The message defined in this clause appears in the help message window when the user presses the Help key from any of the fields in the field list. The Help key may be defined in the OPTIONS statement in the body of the program, by default CONTROL-W is the help key.

The integer expression identifies the message within the help file. The help file which contains the help message must be specified in the HELP FILE clause of the OPTIONS statement that precedes the



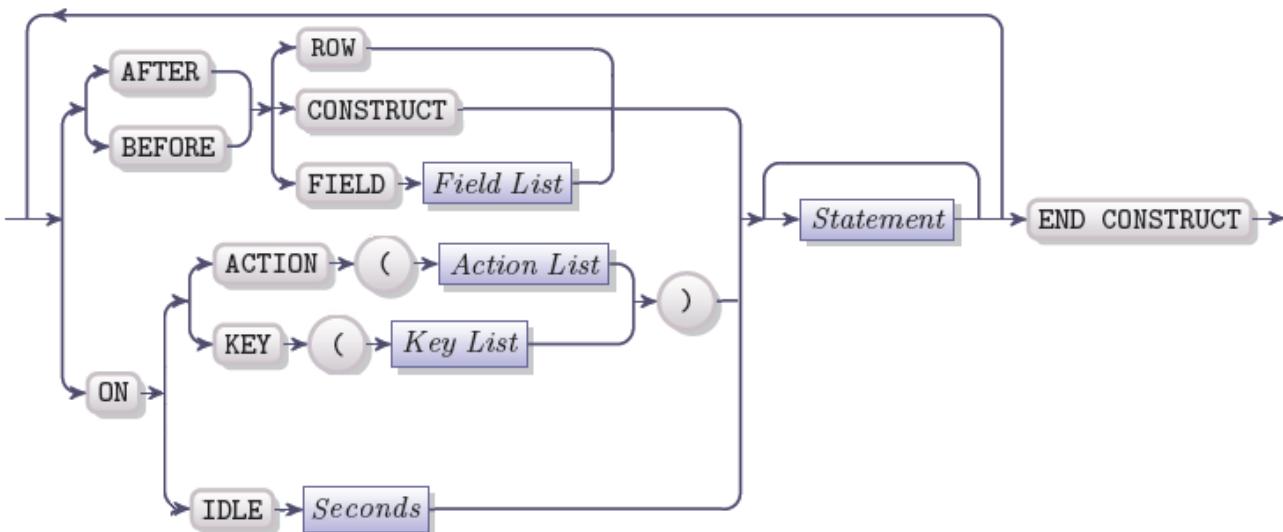
CONSTRUCT statement. If 4GL cannot open the specified help file, if the indicated number is greater than 32,767, or if the number is absent from the help file an error message will appear. The **qmsg** command can be used in the command line environment to create a compiled version of the help file.

	Note: A non-compiled help file is a message file with the extension .msg (help_file.msg). Only a compiled message file can be referenced as a help file. Pay attention that a compiled help file has another extention, e.g. .erm (help_file.erm).
--	---

To provide a specific help message for a specific field, use the OK KEY clause of the CONSTRUCT input control block with the infiel() operator and showhelp() function. If the help is provided for the specified fields, the help messages should be displayed in a 4GL window within the 4GL screen and not in a separate help window.

The CONSTRUCT Input Control Clauses

A CONSTRUCT input control clause is an optional clause that controls the input of the data for the query. If it is included into the CONSTRUCT statement, it must contain at least one statement and an activation clause, which defines when the block is to be executed.





Element	Description
Field List	A list of one or more form fields separated by commas
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks
Key List	A list that consists of one to four key names separated by commas
Statement	The statement block of the control clause
Seconds	An integer value specifying the number of timeout period
END Statement	END CONSTRUCT keywords

The CONSTRUCT input control block is used for the following purposes:

- To specify statements to be executed before and after the query by example
- To specify statements to be executed before and after the given field
- To specify statements to be executed after pressing a specified key or button
- To specify the next field to which the cursor will be moved
- To define when to exit the CONSTRUCT statement

The form is temporally non-active while the statements of the input control block are executed, after that the form is reactivated.

Statement Block

The Statement block can include:

- Any 4GL executable statement
- SQL statements that can be embedded into 4GL code
- NEXT FIELD keywords
- CONTINUE CONSTRUCT keywords
- EXIT CONSTRUCT keywords
- BEGIN ... END logical block (for the details see the "[Logical Blocks](#)" section of this reference)

The Order of Input Control Clauses

The input control clauses can be included in a CONSTRUCT statement in any order. When one or more input control clauses are included into the CONSTRUCT statement, the END CONSTRUCT keywords must be used to mark the end of the CONSTRUCT statement. 4GL executes the clauses in the following sequence without regard to the order in which they are placed within the CONSTRUCT statement:

- BEFORE CONSTRUCT - before any data have been entered
- BEFORE FIELD - before any value is entered into the specified field
- ON KEY and ON ACTION - after the specified key or button has been pressed
- AFTER FIELD – after a value has been entered to the specified field
- AFTER CONSTRUCT – after data entry has been finished

If a CONSTRUCT statement contains no input control clause, the program waits until user enters all the data required and terminates the CONSTRUCT statement on pressing the Accept key. No END CONSTRUCT statement is required.

The BEFORE CONSTRUCT Clause

The BEFORE CONSTRUCT clause specifies the statements that are to be executed before anything is entered into the screen fields. All the fields are cleared at the beginning of the CONSTRUCT statement execution. The BEFORE CONSTRUCT block is typically used to display some default values to the screen fields. A



CONSTRUCT statement can contain only one BEFORE CONSTRUCT clause. To specify the first field which is to become active, you can use the NEXT FIELD clause in the BEFORE CONSTRUCT clause.

```
BEFORE CONSTRUCT
DISPLAY rec_list.* TO scr_rec.*
NEXT FIELD f001
```

The BEFORE FIELD Block

This block specifies a set of actions to be executed before anything is entered to the specified field. The BEFORE FIELD block is executed after the cursor is moved to the field but before it becomes active for input. One field can have only one BEFORE FIELD block. If you use a NEXT FIELD clause within the BEFORE FIELD block the access to the field will be restricted and the user will not be allowed to enter data into this field.

```
BEFORE FIELD city
MESSAGE "Press F3 to display the list of the partner cities."
```

The ON KEY Clauses

The ON KEY control blocks are used to define the actions to be executed when the user presses a specific key combination. This key combination can be assigned to a form widget and the actions are triggered when such widget is pressed. However, if a key combination is not assigned to any widget, it can be activated with the help of the keyboard.

The following keywords can be specified in the Key List:

ESC/ESCAPE	ACCEPT
NEXT/ NEXTPAGE	PREVIOUS/PREVPAGE
INSERT	DELETE
RETURN	TAB
INTERRUPT	HELP
LEFT	RIGHT
UP	DOWN
F1 – F256	CONTROL- <i>char</i>

Any character can be used as *char* in the combination CONTROL-*char* except the following characters: A, D, H, I, J, K, L, M, R, or X.

Some keys require special attention if used in the ON KEY block:

Key	Usage Features
ESC/ESCAPE	If you want to use this key in ON KEY block, you must specify another key as the Accept key in the OPTIONS block, because ESCAPE is the Accept key by default
F3	It is the default Next key, if you want to use it in the ON KEY block, you must specify another key as the Next key in the OPTIONS block
F4	It is the default Previous key, if you want to use it in the ON KEY block, you must specify another key as the Previous key in the OPTIONS block
INTERRUPT	DEFER INTERRUPT statement must be executed in order that this key could be used in the ON KEY block. On pressing the INTERRUPT key the corresponding ON KEY block is executed, int_flag is set to non-zero but the



	CONSTRUCT statement is not terminated
QUIT	DEFER QUIT statement must be executed in order that this key could be used in the ON KEY block. On pressing the QUIT key the corresponding ON KEY block is executed and <code>int_flag</code> is set to non-zero
CTRL-char (A, D, H, L, R, X)	4GL reserves these keys for field editing and they should not be used in the ON KEY block
CTRL-char (I, J, M)	These key combinations by default mean TAB, NEWLINE and RETURN. If they are used in the ON KEY block, they cannot perform their default functions while the OK KEY block is functional. Thus, if you use these keys in the ON KEY block, the period of time during which this block is functional should be restricted.

Some restrictions in key usage might be applied depending on your operational system. Many systems use such key combinations as CONTROL-C, CONTROL-Q, and CONTROL-S for the Interrupt, XON, and XOFF signals.

When an ON KEY block is present within the input control block, 4GL takes the following actions:

- The input in to the current field is temporally suspended.
- The characters entered by the user are preserved in the input buffer.
- The statements included into the corresponding ON KEY block are executed.
- The input buffer is restored for the current field.
- The input in the field is restored with the cursor being at the end of the list of characters extracted from the buffer.

This default set of actions can be changed. The input may be resumed not for the same field but for any field of your choice, if the corresponding ON KEY block contains the NEXT FIELD clause.

```
ON KEY (CONTROL-G)
IF lname IS NOT NULL THEN
NEXT FIELD address
END IF
```

It is also possible to change the value for the corresponding variable preserved in the input buffer and to display the changed value to the field by placing the necessary statements in the ON KEY block.

```
ON KEY (F12)
LET curr_f = "N/A"
```

You can specify up to four keys in one ON KEY clause, pressing any of these keys will result in executing the ON KEY clause. The key names must be enclosed in parentheses and separated by commas.

ON ACTION Clauses

The actions which you have assigned to the buttons in the form specification file can be used in an ON ACTION clause. The ON ACTION clauses are treated by 4GL very much like the ON KEY clauses. 4GL executes the statements specified in it, when the user presses a button or other widget on the form to which the corresponding action is assigned. Whereas an ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard, an ON ACTION clause is activated with the help of a form widget.



An event (action) can be assigned to the following widgets: button, radio button, check box, function field, and hotlink. It must be entered into the corresponding field in the Graphical Form Editor without quotation marks and white spaces. In the text form editor it must be enclosed into quotation marks:

```
CONFIG = "action {Name of the Button}"
```

The actions in an ON ACTION clause must be represented by a character string enclosed both in quotation marks and in parentheses and it must be the same as the name of the action in the form specification file. One ON ACTION clause can contain up to four action names enclosed in quotation marks and separated by commas:

```
ON ACTION ("event1", "event2", "event3")
```

There can be any number of the ON ACTION clauses in a CONSTRUCT statement.

As well as with the ON KEY clause, any 4GL or SQL statements can be included into an ON ACTION clause. The statements of a corresponding ON ACTION clause are executed when the widget within a screen form to which the action is assigned is pressed. The form where the CONSTRUCT statement takes place is deactivated during the statements are executed, though the statement is not terminated and the form will be reactivated, when 4GL finishes executing the ON ACTION clause, unless it contains the EXIT CONSTRUCT statement. If the ON ACTION clause contains the EXIT CONSTRUCT statement, the CONSTRUCT statement will be terminated.

The example below represents an ON ACTION clause which is triggered when the button with the "update" event is pressed.

```
ON ACTION ("update")
CALL update(a,b)
```

The ON IDLE Clauses



Element	Description
Seconds	An integer value specifying the time period in seconds

The ON IDLE clause is used to specify the actions that will be taken if the user performs no actions during the specified period of time.

The *seconds* parameter should be represented by an integer value or a variable which contains such a value. If the value is specified as zero, the input timeout is disabled.

It is advised that you specify the *seconds* parameter as a relatively long period of time (more than 30 seconds), because shorter delays may be caused by some external situations that distract the user from the application, so, the control block will be executed inappropriately:

```
ON IDLE 120
CALL fgl_message_box("You've been out for 2 minutes")
END INPUT
```



The AFTER FIELD Clauses

The corresponding AFTER FIELD clause is executed when the cursor leaves the current field, if this field has an AFTER FIELD clause associated with it. Each field can have only one AFTER FIELD clause associated with it. The cursor can be moved from the current field by pressing any of these keys: any arrow key, RETURN, Accept key, TAB.

If the AFTER FIELD block contains the NEXT FIELD keywords, the cursor will be placed in the field specified as the next one.

	Note: If every field contains the NEXT FIELD clause, the user will not be able to exit the form.
---	---

The users can terminate a CONSTRUCT statement by pressing the Accept, Interrupt or Quit keys at any time; or, after the last form field, using the Tab or Return keys. This default behaviour can, however, be overridden by using the NEXT FIELD keywords in the AFTER FIELD block of the last form field. The same effect will be achieved by specifying the INPUT WRAP keywords in the OPTIONS statement.

```
AFTER FIELD city
MESSAGE "Press F5 to search for the corresponding record"
AFTER FIELD phonecode
NEXT FIELD lname
```

The AFTER CONSTRUCT Clause

Only one AFTER CONSTRUCT clause can be included into the CONSTRUCT statement. This block will be executed, if one of the following keys is pressed: Accept key, Interrupt key or Quit key. For the last two keys DEFER INTERRUPT or DEFER QUIT statements must be in effect, otherwise the program will be terminated immediately and no query will be performed.

The AFTER CONSTRUCT block is executed after the Accept key is pressed but before the Boolean expression for the query by example is created. This block can be used to validate the entered data or to save them. If CONTINUE CONSTRUCT or NEXT FIELD are included in this block, the CONSTRUCT statement will not be terminated and the cursor will return to the form.

```
AFTER CONSTRUCT
IF NOT field_touched(ord_id) AND NOT field_touched(cl_id) THEN
MESSAGE "You must specify either order number or the customer name
to perform the query."
CONTINUE CONSTRUCT
END IF
```



	<p>Note: If CONTINUE CONSTRUCT or NEXT FIELD keywords are not included into a conditional statement when used in the AFTER CONSTRUCT block, the CONSTRUCT statement will not be terminated and the user will not be able to leave the form.</p>
---	--

The AFTER CONSTRUCT clause is not executed, and the CONSTRUCT statement is not completed, if:

- The Interrupt or Quit key is pressed with **no** corresponding DEFER statement in effect – a query will not be performed and the program will be terminated
- 4GL encounters the EXIT CONSTRUCT statement – the program control will be passed to the statements following the END CONSTRUCT statement, but the query by example will not be constructed

In the above listed situations the query cannot be performed, because the CONSTRUCT statement will not create the aggregate search criterion which could be used in a query.

The NEXT FIELD Clause

By default the cursor is moved from field to field in the order specified in the FROM clause, or in the order implied by the ON clause if the variables clause of the CONSTRUCT statement contains the BY NAME clause. The NEXT FIELD clause is used to change the default order.

The following fields are qualified to be used in the NEXT FIELD clause:

- Any field in the current form, its name should follow the NEXT FIELD clause and it should correspond to the name of the field in the ATTRIBUTES section of the form specification file
- The next field, that is after the current one according to the order of the fields defined by the FROM or BY NAME clauses. The NEXT FIELD should be followed by the keyword NEXT.
- The previous field, that is before the current one according to the order of the fields defined by the FROM or BY NAME clauses. The NEXT FIELD should be followed by the keyword PREVIOUS

The NEXT FIELD clause can appear in any of the input control clause blocks, but they are commonly used in the AFTER FIELD, ON KEY, or AFTER CONSTRUCT blocks.

The statements that follow the NEXT FIELD clause are not executed and the cursor is immediately moved in the direction determined by this clause.

```
ON ACTION ("validation")
    IF f005 IS NOT NULL THEN
        NEXT FIELD NEXT
    ELSE
        NEXT FIELD f005
    END IF
```



	Note: As a general rule, you should not use the NEXT FIELD clause to move the cursor through every field in the form. It is advisable that you specify the fields in the correct order in the Variables clause of the CONSTRUCT statement.
---	---

The CONTINUE CONSTRUCT Keywords

The CONTINUE CONSTRUCT statement is used to exit any of the blocks within the input control clause of the CONSTRUCT statement and to return the cursor to the form. All the statements that follow the CONTINUE CONSTRUCT keywords are skipped. It is useful for quitting conditional loops. It can also be used in an AFTER CONSTRUCT block to review the values in field buffers and, dependent on the values, put the cursor back in the form.

When an AFTER CONSTRUCT clause contains an IF test that identifies a field for which the user has to complete an action, the NEXT FIELD keywords should be used rather than CONTINUE CONSTRUCT statement to position the cursor.

The EXIT CONSTRUCT Keywords

The EXIT CONSTRUCT can be placed in any of the input control blocks. It triggers the following set of actions:

1. The statements between the EXIT CONSTRUCT keywords and the END CONSTRUCT statement are skipped
2. The query by example is not constructed
3. The Boolean expression is created and stored in character variable
4. The execution of the program is resumed from the first statement following the END CONSTRUCT statement

If 4GL encounters the EXIT CONSTRUCT keywords, the AFTER CONSTRUCT clause is not executed.

The END CONSTRUCT Keywords

The END CONSTRUCT keywords are optional. They are used only if a CONSTRUCT statement contains at least one input control clause.

The END CONSTRUCT keywords complete the CONSTRUCT statement. They should follow the last block of the input control clause of the CONSTRUCT statement.

The Search Criteria for a Query by Example

Search criteria are specified by the user when they are entered into the fields of the form. It is possible to search data lesser than the entered value, greater than the entered value, etc. To do this the user must enter special symbols to a field. Here is the list of the acceptable symbols:

Symbol	Name	Data Types	Pattern
= or ==	equal to	All	==x, =x, =
>	greater than	All	>x
<	less than	All	<x
>=	not less than	All	>=x
<=	not greater than	All	<=x
<> or !=	not equal to	All	!=x, <>x



: or ..	range from..to	All	x:y, x..y
* Wildcard	replaces any string	CHAR, STRING, VARCHAR	*x, x*, *x*
?	wildcard for a single character	CHAR, STRING, VARCHAR	?x, x?, ?x?, x??
 	logical OR	All	a b...
[]	List of values (see below)	CHAR, STRING, VARCHAR	[xy]*, [xy]?



Note: The '..' range operator is required when you need to use a range operator for DATETIME or INTERVAL data types that include ':' symbols. E.g. 05:12:36.

It is not possible to perform a query by example on BYTE and TEXT, as well as on the formonly fields which are not associated with columns in a database.

The explanation of the symbols in the previous table:

Symbol	Explanation
x	Any value compatible with the data type of the field. It must follow one of the symbols from the previous table without any spaces
= value	The equal sign is the default symbol for non-character fields, and for character fields in which the search value contains no wildcards
=	The equal sign not followed by any value searches for NULL value. If the user wants to find any value that is the same as one of the special search criteria symbols he or she must explicitly enter the equal sign.
>, <, >=, <=, <>	These symbols imply an ordering of the data. E.g. '>1' means 2, 3, 4 and so on, '<C' means 'A' and 'B' as the characters are ordered ascending A to Z. For DATE or DATETIME data types 'greater than' means later and for INTERVAL data type it means a longer period of time.

A query by example cannot combine the described above relational operators with the range, wildcard, or logical operators described below. Any character following a relational operator is interpreted as literal.

The CONSTRUCT statement puts single quotation marks (') at both sides of the values of the following data types: CHAR, DATE, DATETIME, INTERVAL, and VARCHAR. For values of the number data types no quotation marks are used, these data types are: FLOAT, SMALLFLOAT, DECIMAL, MONEY, INTEGER, SMALLINT, and SERIAL.

It is possible to specify a range or to use syntax similar to that of the MATCHES operator:

Symbol	Explanation
: (Colon)	The colon in <i>x: y</i> makes 4GL search for all values between the <i>x</i> and <i>y</i> values, inclusive. In order the expression to be valid the <i>y</i> value must be larger than the <i>x</i> value. If you enter 3: 15 as the search criterion all rows with a value from 3 through 15 in that column will match this



	criterion. For the data of character type, the criterion w: z will result in finding all the columns with values from w through z (that is w, x, y, z) (For DATETIME and INTERVAL data types, the '..' symbol must be used to specify ranges.)
.. (Two periods)	Two periods symbol is a synonym for the colon (:) in DATETIME and INTERVAL ranges to avoid ambiguity with time-unit separators in hh:mm:ss values.
* (Asterisk)	<p>The asterisk is a wildcard that represents zero or more characters. It is used in the following way:</p> <p>If you enter *th as a search criterion, 4GL will search for all the strings that end with th. E.g. "Smith", "Blacksmith".</p> <p>If you enter *S as a search criterion, 4GL will search for all the strings that end with th. E.g. "Smith", "Sunders"</p> <p>If you enter *an* as a search criterion, all strings containing the letters 'an' will match this criterion, with any number of characters preceding these letters and following them. E.g. "Thailand", "Anjou" and "Taiwan" will be found</p>
? (Question mark)	<p>The question mark is a wildcard that represents a single character. It can be used to find values the number of characters in which is fixed:</p> <p>If you enter "Jo??" you will find both "John" and "Joan" but not "Joanna"</p> <p>If you enter "L???man" you will find "Linkman", "Longman" but not "Lawman"</p>
 (Pipe)	The pipe symbol between values represents the logical OR operator. The search criterion "3009 1623 7765 will find the rows with either 3009, 1623 or 7765 in the corresponding column
[] (Brackets)	The brackets delimit a set of values. They can be used with wildcard symbols to enclose the list of characters that serve as search criterion. E.g. [*so?]
^ (Caret)	If the caret is used as the first character within the brackets, it specifies the logical complement of the set. It matches any character that is not included to the list. E.g. if you enter [^AB]* as a search criterion, you will find all strings that begin with any letter except for A and B
- (Hyphen)	A hyphen between characters within brackets specifies a range. E.g. if you enter [^d-f*] as a search criterion, you find all strings beginning with characters other than lowercase d, e, or f. If you omit the wildcards (*) or (?), 4GL treats the brackets as literal characters and will look for the very string [^d-f]

Searching for All Rows

If none of the fields contains search values when the user completes an entry for the CONSTRUCT statement, " 1=1" is used as the Boolean expression. (This string begins with a blank character.) When placed in a WHERE clause of a SELECT statement such expression will result in selecting all the rows of the specified tables. You can test for this expression to see whether any search criteria have been entered and if not to return the program control to the CONSTRUCT statement.

```

CONSTRUCT BY NAME search ON client.*
    IF search = " 1=1" THEN
        MESSAGE "You entered nothing. Please enter search criteria"
        SLEEP 1
    
```



END IF

Positioning the Screen Cursor

Screen cursor is moved from one field to the next one in the order specified by the FROM clause or by the ON clause in case the BY NAME clause with the help of RETURN or TAB. The following arrow keys can be pressed to change the position of the cursor:

Arrow key	Action
↓	DOWN ARROW moves the cursor to the next field. With the FIELD ORDER UNCONSTRAINED option specified in the OPTIONS statement, the cursor is moved to the field below the current field. If there is no field below the current field, the cursor is moved to the field on the right, if any.
↑	UP ARROW moves the cursor to the previous field. With the FIELD ORDER UNCONSTRAINED option specified in the OPTIONS statement, the cursor is moved to the field above the current field. If there is no field below the current field, the cursor is moved to the field on the left, if any.
←	LEFT ARROW moves the cursor one space to the left within the field without erasing or changing the contents of the field. If the cursor is at the beginning of the field, it is moved to the previous field. The effect is the same as of the key CONTROL-H.
→	RIGHT ARROW moves the cursor one space to the right within the field without erasing or changing the contents of the field. If the cursor is at the end of the field, it is moved to the beginning of the next field. The effect is the same as of the key CONTROL-L.

The arrow keys do not influence the data in the fields, they do not change or erase the data. When the cursor is moved to a new field the Error line and Message line are cleared.

If the search criteria exceed the length of the field, the cursor is automatically moved down to the Comment line allowing the user to continue entry. When the user presses RETURN or TAB, the Comment line is cleared, though all the entered criteria are preserved in the field buffer including the part of it that is not visible in the screen form.

Using the WORDWRAP Attribute in the CONSTRUCT Statement

When the data exceeds the length of the form field, 4GL moves the cursor to the overflow line and the user can continue entering data. A form field with the WORDWRAP attribute can span several lines. The cursor can skip some segments or it may be left in arbitrary locations inside the segment, if the segments of a multi-segment field are not aligned in a single column.

You should avoid placing the fields as shown below:

```
[f001] [f001]  
[f001]
```

It is advisable that you place the segments of a multi-segment field in one column:

```
[f001]  
[f001]
```



[f001]

Any values are ignored if they are entered in any segment of a multi-segment field but the first one.

Editing while in a CONSTRUCT Statement

The following keys can be used for editing during the CONSTRUCT statement processing:

Key Combination	Effect
CONTROL-A	Toggles between type-over and insert mode
CONTROL-D	Deletes all the characters beginning with the position of the cursor and to the end of the field
CONTROL-H	Moves the cursor one space to the left within a field achieving the same result as the LEFT ARROW key
CONTROL-L	Moves the cursor one space to the right within a field achieving the same result as the RIGHT ARROW key
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes the character below the cursor

Completing a Query

The CONSTRUCT statement is terminated when one of the following actions is executed:

One of the following keys is pressed: ACCEPT, RETURN, TAB, INTERRUPT or QUIT key
The EXIT CONSTRUCT statement is executed

The user must press the ACCEPT key to complete the query under these conditions:

INPUT WRAP is specified in the OPTIONS section
An AFTER FIELD block for the last field includes a NEXT FIELD clause

ACCEPT, RETURN, TAB, INTERRUPT or QUIT keys terminate the CONSTRUCT statement as well as the query by default.

If a DEFER INTERRUPT statement has been executed previously, pressing Interrupt key will result in the following actions:

The built-in variable **int_flag** will be set to TRUE
The CONSTRUCT statement will be terminated but the program will be still running

If a DEFER QUIT statement has been executed previously, pressing Quit key will result in the following actions:

The built-in variable **quit_flag** will be set to TRUE
The CONSTRUCT statement will be terminated but the program will be still running

In either case the variable that is to preserve the query criteria is set to NULL and it will cause an error if you try to use it in the WHERE clause. Set any non-zero value of int_flag or quit_flag to zero (FALSE) before execution of the CONSTRUCT statement to avoid such error. Unless the CONSTRUCT statement is terminated with the help of the EXIT CONSTRUCT key words, the AFTER CONSTRUCT block will be executed before termination.



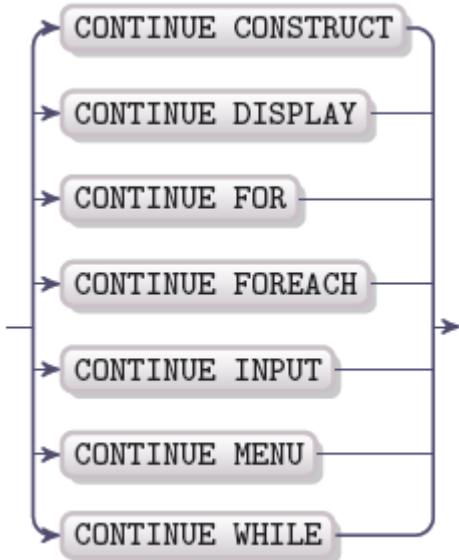
Below is an example of a CONSTRUCT statement used in a query. The variable *var*, which is used in the PREPARE statement, has been previously defined as CHAR(250).

```
CONSTRUCT BY NAME search_6
    ON lname, fname, address, phonecode
    ATTRIBUTE(UNDERLINE)
    LET var = "SELECT * FROM clients ",
    "WHERE search_6 CLIPPED"
    PREPARE s_1 FROM var
    DECLARE cursor_1 CURSOR FOR s_1
    FOREACH cursor_1 INTO client_rec.*
    ...
END FOREACH
```



CONTINUE

The CONTINUE statement is used to cause the FOR, WHILE and several other statements to begin a new cycle, if it is permitted by the present condition. The CONTINUE statement is also used to return from a submenu to the upper menu in the MENU statement. Here are the statements which can be used with the CONTINUE statement:



The CONTINUE INPUT statement is used both for INPUT and INPUT ARRAY statements. The CONTINUE DISPLAY combination is used for the DISPLAY ARRAY statement.

The CONTINUE keyword used in the WHENEVER statement is discussed in detail in the [WHENEVER](#) section of this reference.

The CONTINUE Statement in the CONSTRUCT, INPUT, and INPUT ARRAY Compound Statements

After CONTINUE CONSTRUCT and CONTINUE INPUT statements all subsequent statements in the current control block are skipped. The cursor is moved to the field that has been active before that; the user is allowed to enter other data to that field. For more information see the sections on the [INPUT](#) and [CONSTRUCT](#) statements in this manual.

The CONTINUE DISPLAY Statement

The CONTINUE DISPLAY keywords can be used only within the DISPLAY ARRAY statement. They are not valid within a simple DISPLAY statement.

After CONTINUE DISPLAY keywords all subsequent statements in the current control block of the DISPLAY ARRAY statement are skipped. The array is redisplayed. However, if one of the previous control blocks has displayed other values to one or more fields of the screen array, these values will continue being displayed



and the initial values will not be restored. For more information see the sections on the [DISPLAY ARRAY](#) statement.

The CONTINUE Statement in the FOR, FOREACH, and WHILE Loops

The CONTINUE FOR, CONTINUE FOREACH, or CONTINUE WHILE keywords start a new cycle of the FOR, FOREACH, or WHILE loop respectively, if conditions allow. If conditions prevent such behaviour, a loop is terminated. For more information see the sections on the FOR, FOREACH and [WHILE](#) statements in this manual.

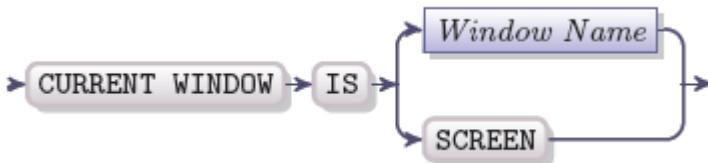
The CONTINUE MENU Keywords

If the CONTINUE MENU statement occurs within MENU compound statement, it causes 4GL to ignore the statements that follow CONTINUE MENU statement and redisplay the menu so that it is possible to choose other menu option. For more information see the section on the [MENU](#) statement in this manual.



CURRENT WINDOW

To make any open 4GL window the current window, use the CURRENT WINDOW statement.



Element	Description
Window Name	The name of a window previously opened with the help of the OPEN WINDOW statement

All 4GL windows within the 4GL screen are stored as a stack which determines the order of windows and the rules according to which a window becomes the current window. When a new 4GL window is created, it becomes the current window and is put at the top of the stack. A 4GL window is removed from the stack only when it is closed, and the window which remains at the top of the stack becomes the current window.

The current 4GL window covers all the other windows and other information that is located under it. It is always fully visible and cannot be covered by any other element of a 4GL program. When a window is declared as the CURRENT WINDOW, it is moved to the top of the window stack, the window which has been current before is located right under it in the stack. To make a window the current window use the statement CURRENT WINDOW followed by the obligatory keyword IS and by the identifier of a window.

The Window Identifier

The CURRENT WINDOW IS keywords must be followed by the identifier of a window which you want to make the current window. The window identifier can be specified as:

- A valid Window Name – provides the CURRENT WINDOW statement with a static window identifier which cannot be changed throughout the program
- The SCREEN keyword – makes the 4GL screen the current window

The input and output can be performed only for the current window. If during the input another form becomes displayed (e.g. by means of an ON KEY clause), the window that contains this form becomes the current window and the input can be performed only using this window. The statements that perform input or output and depend on the current window are: CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, and MENU. When one of these statements resumes, the original window of the statement becomes the current window. A window remains the current 4GL window until another window is declared as the current one with the help of the CURRENT WINDOW statement or until it is closed by the CLOSE WINDOW statement.

The Window Name

The window name is static, if it directly follows the CURRENT WINDOW keywords. It can be preceded with optional scope indicators and has the following syntax:

The example below illustrates a static CURRENT WINDOW statement where the window name is a constant:



```
CURRENT WINDOW IS my_window
```

If you specify a window that contains a screen form as the current window, the form automatically becomes the current form.

The SCREEN Keyword

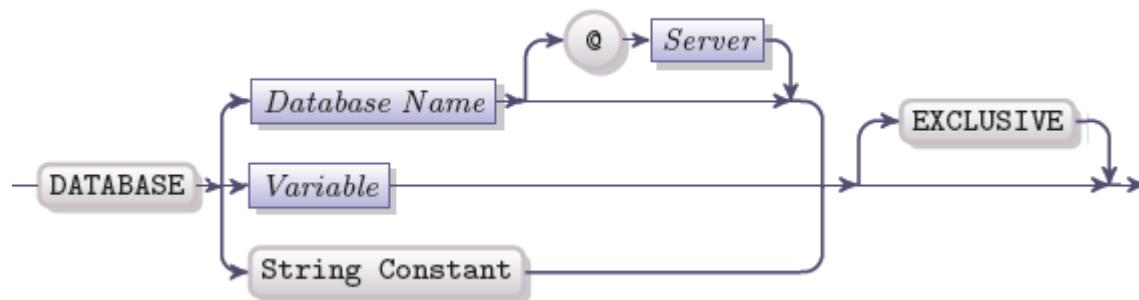
When a 4GL program is run, the 4GL screen becomes the current window. To make the 4GL screen the current window during the runtime of a program after one or more 4GL windows have been placed over it, use the keyword SCREEN instead of the name of a window:

```
CURRENT WINDOW IS SCREEN
```



DATABASE

Use the DATABASE statement to open the database which will become the default database and the current database.



Element	Description
Database Name	The name of the database
Server	The name of the host system where database resides.
Variable	A variable that contains the database specification

The DATABASE statement is required, if your 4GL program references a database at least once. This statement can be used in two ways:

- You can specify the current database during the runtime so that the statements could access the data in the database at runtime.
- You can specify the default database which will be opened automatically when the program is run and which will be used during compilation (it also allows the INITIALIZE and VALIDATE statements to access syscolatt or syscolval)

	Note: The DATABASE statement should be used if the application intends to connect to one database at a time. For setting multiple database connections see the CONNECT TO statement description.
--	---

The Specification of a Database

The DATABASE statement specifies any accessible database among those that are supported by Querix tools (Informix, Oracle, MySQL, PostgreSQL, SQL Server and others). The database specified in the DATABASE statement becomes the default database during compilation and the current database at runtime. To change the database during runtime you must include the CLOSE DATABASE STATEMENT followed by another DATABASE statement specifying another database. You can also use table qualifiers if you want to use different database for some purposes but do not want to change the current database. (for detailed information on the table qualifiers see the chapter on 4GL Expressions, "[Table Qualifiers](#)" section).



If you want to use another database on the same server as the current database, you do not need to use the CLOSE DATABASE statement. If you use another DATABASE statement the previous database will be closed and the new database will be opened. The CLOSE DATABASE statement is required before opening another database if it is located on a different server.

If you specify a database which 4GL cannot open or cannot find, a compile-time error will appear.

The database identifier is recognized by 4GL, if the database resides in your current directory or if the path to it is specified in the DBPATH variable. Otherwise, you need to follow the DATABASE statement with the complete path-name of the database or by the variable that contains the complete path to it.

When the database is opened, the locale consistency check takes place. If the locales of the database and your system do not match, the request for database service is rejected. The check is performed by means of transmitting the locale categories for COLLATION and CTYPE from the system to the database, they are compared with these categories of the database.

The COLLATION and CTYPE categories are stored in the database system table, they are recorded when the database is created and cannot be changed.

The Default Database

If you have the DEFINE statement that uses the LIKE keyword to specify that a record or a variable has the same data types as a column or columns in a database in your 4GL code, you need to include the DATABASE statement to specify a default database at compile time.

To specify the default compile-time database the DATABASE statement must precede the first DEFINE... LIKE, INITIALIZE... LIKE or VALIDATE... LIKE statement in the source code, it must also precede the GLOBALS section. The database identifier must be expressed explicitly; it must not be a variable. The keyword EXCLUSIVE is not valid when you declare a default database.

You may repeat the DATABASE statement in every program block in which the database will be referenced. It is also possible to include the DATABASE statement before the GLOBALS section of the GLOBALS file, which will be referenced at the beginning of every program block that uses the global variables. (See the details about the usage of the GLOBALS file in the "[GLOBALS](#)" section of this chapter) In this case, the GLOBALS file will have the following structure:

```
DATABASE database_name
GLOBALS
...
END GLOBALS
```

You can have as many DATABASE statements as required outside of the program's MAIN block or other functions. The currently active database for object type determination is the result of the last global DATABASE statement the program has processed (either in the main program body, or in a GLOBALS statement).

The default database a program will connect to is determined by the last DATABASE statement in the file containing a MAIN block.



The Current Database

If your 4GL application interacts with the database by means of SELECT, INSERT and other SQL statements, but you do not have the LIKE keywords in the declaration blocks, you must specify the current database which can be referenced at runtime. The DATABASE statement must occur within a FUNCTION block or the MAIN block, it must follow the DEFINE statements of the corresponding blocks. You can use a variable as the database identifier and you can use the keyword EXCLUSIVE.

If no DATABASE statement follows the MAIN block or a FUNCTION block and if a program block includes the DATABASE statement that specifies the default database, the default database will be also automatically used as the current database.

No DATABASE statement can be included into the REPORT program block. It cannot also be included into the PREPARE statement.

The EXCLUSIVE keyword

If a current database is opened with the EXCLUSIVE keyword, it will prevent any other user from accessing this database except the user that opened the database in the exclusive mode. This keyword can occur only when the DATABASE statement is used within the MAIN or FUNCTION block. You cannot change the exclusive mode of the database while it is open. To allow other users to access to the database you must close the database opened in the exclusive mode by means of the CLOSE DATABASE statement. You can also reuse the DATABASE statement without the EXCLUSIVE keyword which will close the database and reopen it in normal mode.

If another user has already opened this database in exclusive mode, you will receive an error if you try to connect to it. If you try to open a database in exclusive mode, though this database is already opened by another user in normal mode, an error will occur and the access will be denied.

Testing the sqlca.sqlawarn Built-in Record

The type of the database opened by the DATABASE statement can be determined after the DATABASE statement has been successfully executed by examining the sqlca.sqlawarn built-in record

- The second element of the sqlca.sqlawarn built-in record contains value "W", if the database uses transactions.
- The third element of the sqlca.sqlawarn built-in record contains value "W", if the database is ANSI-compliant.
- The fourth element of the sqlca.sqlawarn built-in record contains value "W", if the database is the IBM Informix Dynamic Server.

The Errors Handling Depending on the Database Type

The type of the default database influences the runtime error handling. If a default database is specified with the help of the DATABASE statement, the runtime error handling depends on the fact whether the default database is ANSI-compliant.

The behaviour of the errors depends on the specific SQL statement used rather than on the ANSI-compliant status of the database at runtime. If a program is compiled against a not ANSI-compliant database and then run against an ANSI-compliant database, the errors will behave as they behave in a non-ANSI-compliant database.



When the ANSI-compliant behaviour is requested with no WHENEVER ERROR CONTINUE statement in effect, ANSI compliance will be in effect if one of the following conditions is true:

- the default database is ANSI-compliant
- the –ansi compilation flag is specified when compiling in console mode
- the DBANSIWARN variable is set

The non-ANSI-compliant method with no WHENEVER ERROR CONTINUE statement will take place if:

- the –anyerr compilation flag is specified when compiling in console mode (STOP is the default action)
- the –anyerr compilation flag is not specified (the default action after expression or data type conversion errors is CONTINUE, in other cases the default action is STOP)

If one part of an application is compiled in a non-ANSI-compliant database and the other in ANSI-compliant one, different parts will have different error handling.

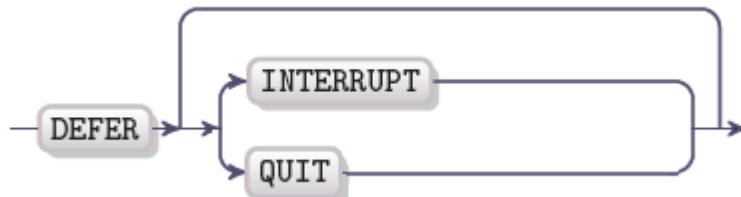
There are several additional issues that concern database connections:

- You can include CONNECT, SET CONNECT and DISCONNECT SQL statements in a 4GL application, they cannot be prepared.
- To connect a 32-bit 4GL client to a 64-bit database server, use network connection and not the shared-memory connection



DEFER

To prevent a 4GL application from termination while executing, if the Interrupt key or Quit key is pressed, use the DEFER statement.



The DEFER statement can be followed either by the INTERRUPT keyword (to prevent a program from termination on pressing the Interrupt key) or by QUIT keyword (to prevent a program from termination on pressing the Quit key). A program that contains no DEFER statement will be terminated if either of these keys is pressed. By default, the Interrupt key is CONTROL-C and the Quit key is CONTROL-\.

If the source code contains a DEFER statement, either the int_flag or quit_flag built-in variable is set to a non-zero value. The int_flag is set to TRUE, if the DEFER INTERRUPT statement is in effect. The quit_flag is set to TRUE when the DEFER QUIT statement is in effect.

The DEFER statement can appear only within the MAIN block and only once in a program. This statement remains in effect after it has been executed, the default actions for the Quit and Interrupt keys cannot be restored. The code which checks the status of the int_flag and quit_flag variables can be included into a program.

Querix offers an alternative to the DEFER INTERRUPT and DEFER QUIT commands. They now can be handled using the global event handlers. For more details, see the WHENEVER statement section of this document.

The Interruption of the Screen Interaction Statements

Screen interaction statements are CONSTRUCT, DISPLAY ARRAY, INPUT, INPUT ARRAY, MENU, and PROMPT. If no DEFER statement has been executed, the Quit and Interrupt keys terminate the execution of these statements and the program itself.

If a DEFER INTERRUPT statement has been executed, the Interrupt key can have other functions within the source code. It can be used as a key in an ON KEY clause. If the Interrupt key is pressed and it is not specified as a key for the ON KEY block, the control is passed to the AFTER INPUT or AFTER CONSTRUCT clause. The AFTER FIELD clause is ignored, if any, the int_flag is set to TRUE. The Quit key has no effect on the screen interaction statements if pressed, though the quit_flag is also set to TRUE.

To be sure that the int_flag or quit_flag has been reset after the corresponding keys have been pressed, you can reset them to FALSE before the screen interaction statements. If the DEFER INTERRUPT statement has been executed, and the Interrupt key is pressed while the PROMPT or DISPLAY ARRAY statement is executed, the program control is passed to the statement following these statements and the int_flag is set to a non-zero value. When the DEFER INTERRUPT statement has been executed, and the Interrupt key is pressed while the MENU statement is executed, the control remains within the MENU statement.



You can specify any other action for the Interrupt key with the help of the ON key clause. The following code checks whether the user has pressed the Interrupt key and defines what actions will be performed if it has been pressed. This is possible only if the DEFER INTERRUPT statement has been previously executed.

```
MAIN
DEFER INTERRUPT
.....
LET int_flag=FALSE
INPUT my_rec.* FROM scr_rec.*
IF int_flag=TRUE THEN
MESSAGE "You have interrupted the input."
END IF
...
END MAIN
```

The INPUT statement has been interrupted, but the program will be executed further, if the user presses the Interrupt key and consequently the int_flag is set to TRUE (non-zero value). The LET statement has previously set int_flag to FALSE to make sure that pressing the Interrupt key during the input will set the int_flag to TRUE.

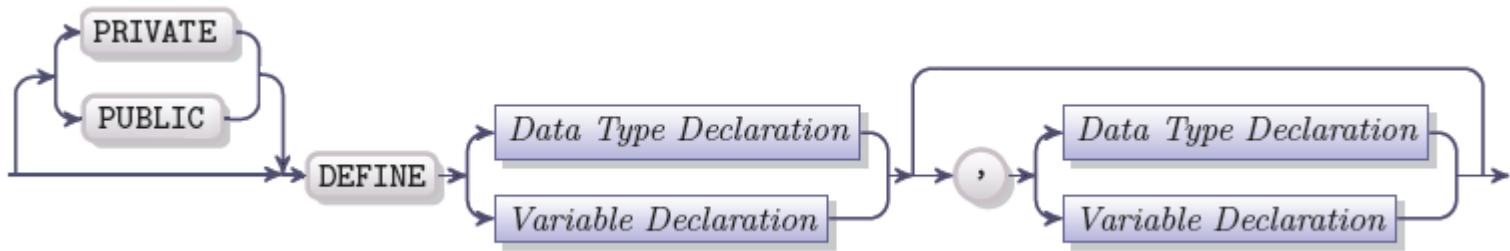
	Note: The int_flag and quit_flag are not reset automatically after the Interrupt or the Quit key has been pressed.
---	---

To make the Interrupt key interrupt SQL statements, your program must include the DEFER INTERRUPT statement and the OPTIONS statements which contains the SQL INTERRUPT ON clause. Otherwise the SQL statements are uninterruptable. To restore the default uninterruptable behaviour of the SQL statements, use the OPTIONS statement with the SQL INTERRUPT OFF clause.



DEFINE

The DEFINE statement is used to declare the names and the data types of the 4GL variables. The DEFINE statement can be followed by one or more declare blocks separated by commas:



Element	Description
Data Type Declaration	A block where a user-defined data type is declared
Variable Declaration	A block where the data type of a variable is declared

There are two patterns of declaration in Querix 4GL:

- Variable Declaration block is used to declare a variable of any 4GL data type or of a user-defined data type (see the "[Variable Declaration](#)" section below)
- Data Type Declaration block is where a new user-defined data type can be declared (see the "[Data Type Declaration](#)" section below)

These blocks can co-exist within one DEFINE statement.

A name of a variable can be any set of characters except for the names that are reserved for the built-in global variables like status, int_flag, quit_flag, etc. A variable that has not been defined cannot be used in the program code.

The total space that can be occupied by the variable names of one program is limited to 2 gigabytes, although your system can impose additional constraints on the storage space available for the variable names. In programs compiled to p-code, the storage space allocated for the variable names of a report or a function cannot be larger than 32,767 bytes.

The variables defined within a GLOBALS file can be visible to all the program modules which use this GLOBALS file.

The Declaration Context

The place within the source code where a variable should be declared depends on the scope of reference of the variable. All the locations where this variable cannot be used are considered to be out of the scope of the variable reference, the locations in which the variable can be referenced are considered to be in the scope of the variable identifier.

The context of the variable declaration also defines whether the storage space for the variable is allocated when the program is loaded to run, or already at the runtime.

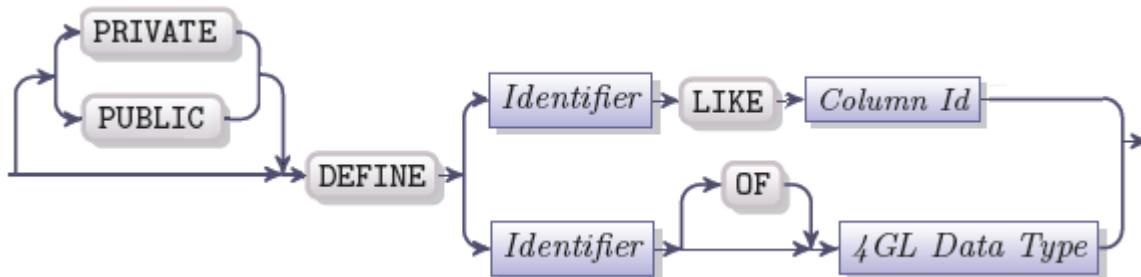


The scope of the identifier and the time when the storage is allocated for it depend on the place in the source code where the variable is declared, that is on the declaration context.

- **Spot Variables** are the variables declared within CASE, IF, WHILE, FOR, or FOREACH statements and the variable declared within the BEGIN...END block located anywhere in the source code. The storage is allocated to them at the runtime when the execution of the corresponding statement begins. Their scope of reference is restricted to the statement or the BEGIN...END block where they are declared and they cannot be used outside it.
- **Local variables** are the variables declared within the MAIN, FUNCTION or REPORT program block. The storage is allocated to them at the runtime when the execution of the corresponding program block begins. Each time a block is entered, a new copy of its local variable is created. Their scope of reference is restricted to the program block where they are declared and they cannot be used outside it.
- **Module variables** are the variables declared outside the MAIN, FUNCTION or REPORT block. To be valid, the declaration of the module variables should appear before the first program block of the program module. The storage is allocated for these variables statically, in the executable image of the application. Their scope of reference is restricted to the module in which they are declared.
- **Global variables** are the variables declared within a GLOBALS block or in a GLOBALS file. They are discussed in detail further in this chapter in the [GLOBALS statement section](#).

Variable Declaration

The variables of any data type or declared LIKE a table column are defined using the following pattern:



Element	Description
Identifier	The name of the declared variable
Column Id	A name of a database column in the format <i>table.column</i> . It can also be preceded by other table qualifiers: as table owner, server name and database
4GL Data Type	Any 4GL data type or user defined data type or a table column preceded by the LIKE keyword

Here are the examples of data types declaration:

- A 4GL data type after the name of the variable

```
DEFINE var1 INTEGER
```

- A previously defined user data type after the name of the variable (for more information about user-defined data types, see the "[Data Type Declaration](#)" section):

```
DEFINE var3 my_datatype
```



The OF keyword is optional, it can be used to specify both 4GL and user-defined data types. Its presence or absence does not influence the result of declaration. The examples below will have the same effect as the examples above.

```
DEFINE var1 OF INTEGER  
DEFINE var3 OF my_datatype
```

You can define a variable of the following Querix 4GL data types:

- Simple Data Type (to see which data types refer to simple, see “Simple Data Types” section of the “Querix 4GL Data Types” chapter)
- Large Data Type (to see which data types refer to large, see “Large Data Types” section of the “Querix 4GL Data Types” chapter)
- Structured Data Type (to see which data types refer to structured, see “Structured Data Types” section of the “Querix 4GL Data Types” chapter)

You can find the information about data types available in Querix 4GL in the “[Data Types of Querix 4GL](#)” chapter.

The simple and large data types are declared by putting the name of the data type after the variable name as shown below:

```
DEFINE  
    var1 CHAR(20),  
    var2 INT,  
    var3 BYTE
```

Those data types that have size or precision can have them in the parentheses following the name of the data type directly (e.g. CHAR(20)). The declaration of structured data types is performed in accordance with declaration patterns illustrated below.

The LIKE Keyword

The keyword LIKE can be used after a variable identifier to define a data type of a variable indirectly. This means that you do not specify the data type of a variable explicitly, but you specify that the data type of this variable is the same as the data type of a table column used in the LIKE clause.

```
DEFINE cl_name LIKE customer.fname
```

The *customer.fname* column has been declared as CHAR(20), the variable *cl_name* will have the same data type: CHAR(20). The owner qualifier must be specified before the table name for a table within a LIKE



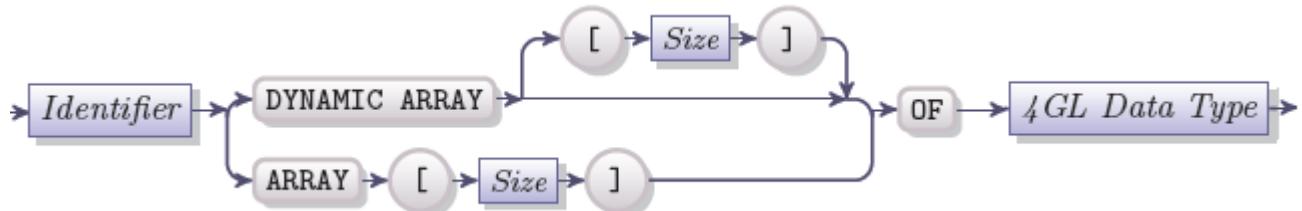
clause, if the default database is ANSI-compliant and you are not the owner of the table, or if the table name is not unique.

You must remember that to define a variable LIKE a database column you need to specify the default database with the help of the DATABASE statement before the first DEFINE... LIKE statement. For detailed information on the default database specification see "Default Database" section in this chapter.

	Note: If the database column specified in the LIKE clause is of SERIAL type, the declared variable will be of INTEGER data type.
---	---

The Program Arrays Declaration

Static and dynamic program arrays are declared in the following manner:



Element	Description
Identifier	The name of the declared variable
Size	The upper bounds for up to three dimensions. Can include from 1 to 3 4GL expressions separated by commas that return positive integers
4GL Data Type	A Querix 4GL or user defined data type. It cannot be ARRAY and DYNAMIC ARRAY data types.

In the declaration of an array data type the ARRAY or DYNAMIC ARRAY keywords are required. An array specifies one-, two- or three-dimensional array which elements are of one data type. A static program array can be of simple data type, of RECORD data type, of large data type, or of the user defined data type (which is discussed further in this chapter). A dynamic array can be of any Querix 4GL data type except ARRAY and DYNAMIC ARRAY data types. It can also be of a user-defined data type. Here are several examples of various array declarations both static and dynamic:

```
DEFINE simp_array ARRAY[1000] OF VARCHAR(200)
```

```
DEFINE dyn_arr DYNAMIC ARRAY OF INTEGER
```

```
DEFINE rec_array ARRAY [500] OF RECORD
    member1 LIKE my_table.col1,
    member2 LIKE my_table.col3,
    member3 INT
END RECORD
```



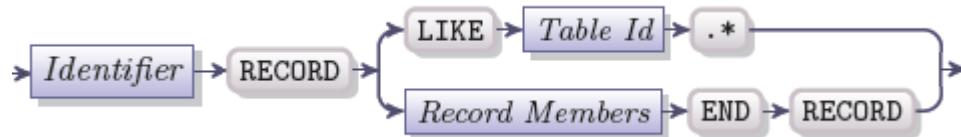
```
DEFINE user_defined_type_array DYNAMIC ARRAY [75] OF my_dtype
```

You cannot define an array (both static and dynamic) of the ARRAY or DYNAMIC ARRAY data type. The ARRAY and DYNAMIC ARRAY data types cannot be specified as an argument of a 4GL function or as its returned value. The CALL and RUN statements cannot contain elements of a static or dynamic array in the RETURNING clauses. Array elements and RECORDS that have arrays as members cannot be used to declare formal arguments within the DEFINE section of a REPORT statement. Non-formal arguments do not share this restriction.

For more information about array data types see the [ARRAY](#) and [DYNAMIC ARRAY](#) sections of this reference.

The RECORD Declaration

A RECORD variable contains other variables as its members. The record members can be of any Querix 4GL data type including the user defined data type; they also can be defined with the LIKE clause.



Element	Description
Identifier	The name of the program record
Table Id	The name of a database table. The identifier can include other table qualifiers, if required
Record Member	A record member

A record member has the same structure as the general variable declaration (see the "[Variable Declaration](#)" section above). A member can be of any data type including structured and large data types. Thus you can have a record member of RECORD data type which will create a nested record.

If a RECORD contains a member defined with a LIKE keyword, the default database must be specified by means of the DATABASE keyword. For detailed information about the default database specification, see the "[Default Database](#)" section in this chapter. A RECORD declaration requires END RECORD keywords at the end, unless it is declared LIKE a table in a database.

The LIKE clause of a record member declaration can contain table qualifiers before the table names. To specify a record that corresponds to all the columns in a table you may use the *LIKE record.** notation (the columns of SERIAL data type will be replaced with the record members of the INTEGER data type):

```
DEFINE cust_rec RECORD LIKE customers.*
```

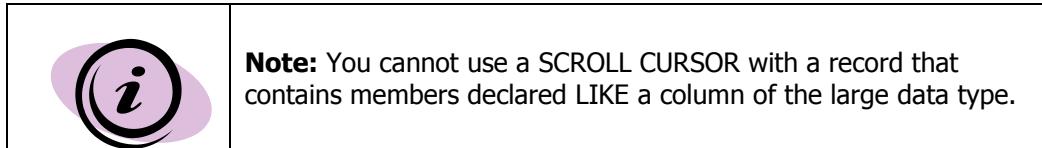
In other cases the END RECORD keywords are required:

```
DEFINE inf_rec RECORD
    memb LIKE inform.cust_id,
    memb2 DATE,
```



```
memb3 INTERVAL
END RECORD
```

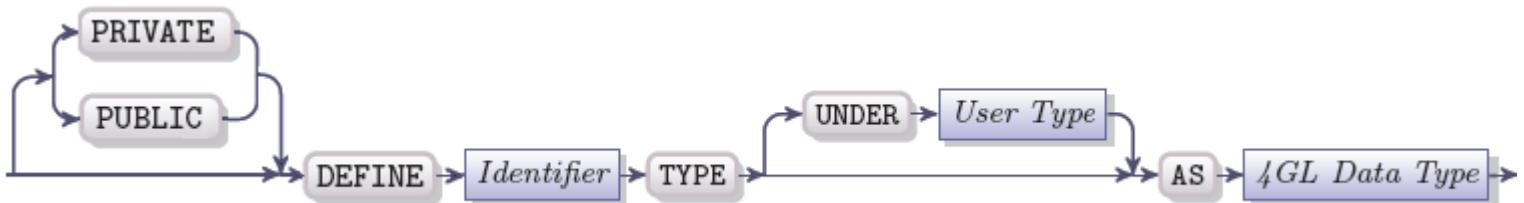
You can declare one record as a member of another record.



For more information about record data type see the [RECORD](#) section of this reference.

Data Type Declaration

In Querix 4GL, it is possible to create your own data type which may be of any of the existing 4GL data types. After being declared, the name of your own data can be used in declaration of program variables as described in the "Variable Declaration" section above. A new data type is declared in the following manner:



Element	Description
Identifier	The name of the record member
User Type	Another user-defined data type which has been declared before the current one
4GL Data Type	Any Querix 4GL or user-defined data type or a table column preceded by the LIKE keyword

- A new data type can be defined as simple data type, e.g. of VARCHAR data type:

```
DEFINE my_sm_type TYPE AS VARCHAR (150)
```

- It can be defined as a structured data type. E.g. the new data type defined as RECORD data type:

```
DEFINE my_st_type TYPE AS
RECORD
    var1 LIKE table.col1
    var2 LIKE table.col2
    var3 CHAR(20)
END RECORD
```

- A new data type can be defined as an ARRAY or a DYNAMIC ARRAY of a simple data type:

```
DEFINE my_ar_rec TYPE AS ARRAY[100]OF CHAR(50)
```



- A new data type can be defined as an array (both static and dynamic) of RECORD data type:

```
DEFINE my_ar_rec2 TYPE AS ARRAY[100]OF
RECORD
var5 INT,
var6 SMALLINT
END RECORD
```

A user defined data type allows you to use this data type for definition of different variables within the source code. It will free you from the necessity to specify the list of record members for every individual variable declaration. If you need to change the data type of all these variables in future, you will not need to change the data type of every individual variable, all you will need is to change the declaration of your data type in the DEFINE ... TYPE AS statement.

The UNDER Keyword

You can add an optional UNDER keyword to modify a previously declared user-defined data type and to use it as another new data type. You can add new members to the user-defined data types declared as records, though you cannot remove the members which have been already added with another declaration.

	Note: The User Type and 4GL Data Type must be of the same data type. E.g. you cannot declare a CHARACTER data type UNDER a user-defined RECORD data type.
--	--

To modify one of your data types, declare a new data type and include the UNDER keyword as in the example below.

```
DEFINE
my_data_type TYPE AS RECORD
    memb1 INT,
    memb2 INT,
    memb3 INT
END RECORD,
modified_data_type TYPE UNDER my_data_type AS RECORD
    memb4 INT
END RECORD
```

In the example above, the "my_data_type" user-defined data type has three record members and serves as the basis for the "modified_data_type" user-defined data type. Thus the "modified_data_type" will have four members: three members of the "my_data_type" and one new member "memb4". If you use the UNDER keyword, you do not have to copy the contents of one user-defined to create another one.



PRIVATE and PUBLIC keywords

The PUBLIC and PRIVATE keywords are valid only for module declarations of variables and user data types. They cannot be used in global or local declarations, where they will cause compilation error.

By default a module variable or a programmer defined data type are private, which means they are visible only inside the module where they are declared. To declare a private variable you may use the PRIVATE keyword or use the DEFINE statement without any preceding keyword.

The private variables and programmer defined data types are not visible to the other modules of the same program, they are also not visible, if the module is imported using the [IMPORT](#) statement.

To make a public declaration, use the PUBLIC keyword before the DEFINE statement. Public module variables and data types are visible from within other modules of the same program and can be visible, if the module is imported using the IMPORT statement.

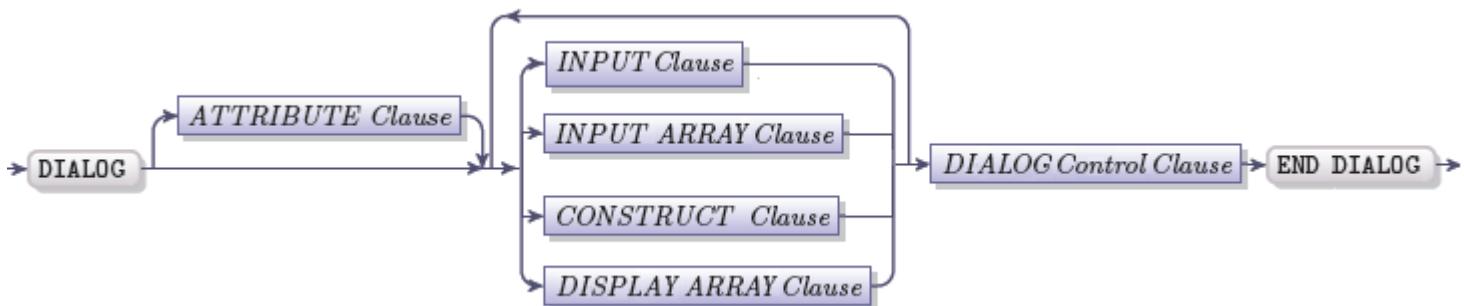
Here are some examples of public declarations:

```
PUBLIC DEFINE      var INT,  
                  rec RECORD  
                      mem1 STRING,  
                      mem2 DATE  
END RECORD  
  
DEFINE st_var STRING --not visible outside the module  
PRIVATE DEFINE p_var MONEY -- not visible outside the module  
  
PUBLIC DEFINE mytype TYPE AS RECORD  
                      m1, m2, m3 STRING  
END RECORD  
FUNCTION func()  
...
```



DIALOG

The DIALOG statement is used to combine different user interaction statements and to execute them simultaneously. It does not in itself allow the user to enter values into fields or view the displayed data, it only serves as a container for input and display clauses and coordinates their work and allows them and work in parallel. Here is its syntax:



Element	Description
ATTRIBUTE Clause	An optional ATTRIBUTE clause where the attributes for DIALOG statement can be specified.
INPUT Clause	A variant of an INPUT statement with its optional clauses and the obligatory END INPUT keywords.
INPUT ARRAY	A variant of an INPUT ARRAY statement with its optional clauses and the obligatory END INPUT keywords.
CONSTRUCT	A variant of a CONSTRUCT statement with its optional clauses and the obligatory END CONSTRUCT keywords.
DISPLAY ARRAY	A variant of a DISPLAY ARRAY statement with its optional clauses and the obligatory END DISPLAY keywords.
DIALOG Control Clause	This clause contains all optional control blocks allowed within the DIALOG statement

The DIALOG statement is used to handle different parts of the same form simultaneously by means by several input or display statements. It cannot include input or display statements which reference fields of a form that is not located in the current window. It also cannot reference fields that belong to different forms. When the DIALOG statement starts executing, it activates the current form which is the most recently displayed form of the current window. This means that it activates all the fields and widgets on the form that are referenced by the underlying sub-dialog clauses. After the DIALOG statement finishes executing, it deactivates the form.

The INPUT, INPUT ARRAY, CONSTRUCT, and DISPLAY ARRAY clauses included into the DIALOG statement share their syntax and behaviour with the independent [INPUT](#), [INPUT ARRAY](#), [CONSTRUCT](#), and [DISPLAY ARRAY](#) statements. However, as a part of the DIALOG statement they have some peculiarities in syntax that will be discussed in detail later in this chapter.

A DIALOG statement can be nested into another DIALOG statement by means of the DIALOG Control Clauses. In this case the parent dialog will be suspended until the child dialog is finished or terminated.



The Dialog ATTRIBUTE Clause

The DIALOG statement can have an optional ATTRIBUTE clause which can have only two attributes: UNBUFFERED and FIELD ORDER FORM.

FIELD ORDER FORM

If this attribute is absent, the cursor moves through the fields in the order in which they are specified in the corresponding INPUT, CONSTRUCT or INPUT ARRAY statement. If it is present, the field order is defined by the TABINDEX attribute value of each field. For more information about the form attributes see "Lycia Development Guide".

```
DIALOG ATTRIBUTE (FIELD ORDER FORM)
...
END DIALOG
```

UNBUFFERED

This attribute indicates that the DIALOG must be sensitive to the program variable changes. The UNBUFFERED attribute can have an optional Boolean value, e.g.: UNBUFFERED = 1 or UNBUFFERED = 0. The TRUE (1) value is equal to the keyword UNBUFFERED used by itself, the FALSE(0) value is equal to omitting the attribute.

```
DIALOG ATTRIBUTE (UNBUFFERED)
...
END DIALOG
```

The UNBUFFERED dialog mode allows you to avoid having to update the field buffer using the built-in functions before being able to use the value of the current field in a control clause. If the UNBUFFERED attribute is present or set to TRUE, the following actions are undertaken by the program, if any ON ACTION or ON KEY block is triggered:

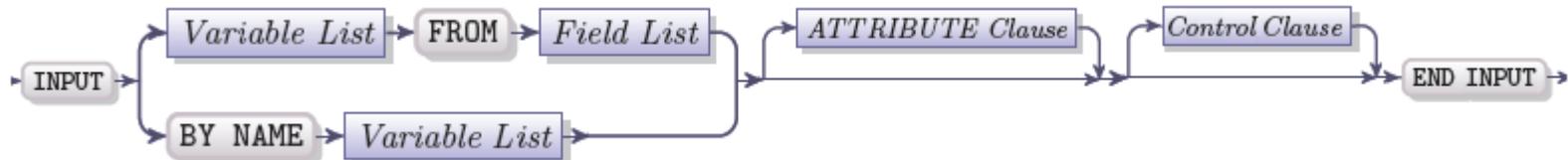
- The value of the current field is written to the underlying variable.
- The statements in the ON ACTION or ON KEY clauses are executed.

If the UNBUFFERED attribute is absent or set to FALSE, the following actions sequence executed by the program, if any ON ACTION or ON KEY block is triggered:

- The value of the current field is written to the field buffer.
- The statements in the ON ACTION or ON KEY clauses are executed.
- The value in the current field is restored from the buffer and the input continues in the same field unless the previously executed clause moved the cursor to another field (i.e. with NEXT FIELD keywords).

The INPUT Clause

The general syntax of the INPUT clause is similar to the syntax of the independent INPUT statement with minor differences. For the general description of the INPUT behaviour and purpose refer to the [INPUT](#) chapter.

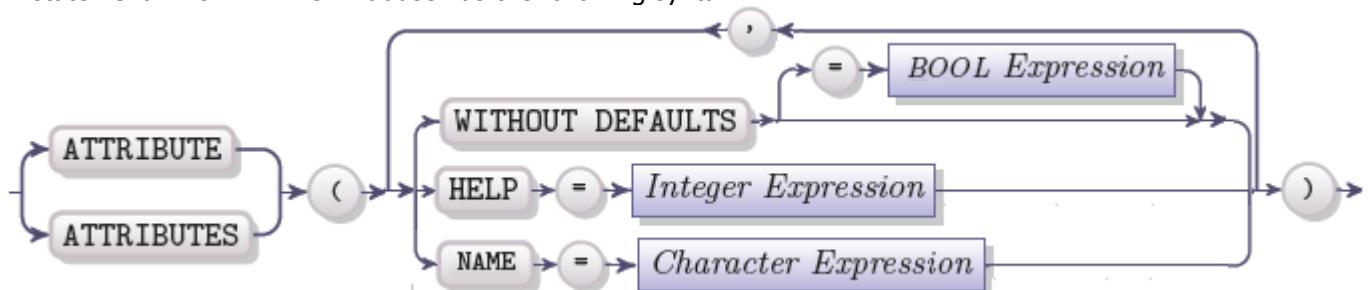


Element	Description
Variable List	The list of variables that will store the entered values
Field List	The list of fields that are bound to the variables
ATTRIBUTE Clause	The list of attributes of the INPUT clause. It is a reduced version of the attributes of the standalone INPUT statement.
Control Clause	INPUT control clauses.

The behaviour of the Binding clause is the same as for the independent INPUT statement. However, the INPUT Clause does not have either HELP or WITHOUT DEFAULTS block and puts some restrictions on the attributes in the ATTRIBUTE clause.

The ATTRIBUTE Clause

The ATTRIBUTE clause of the INPUT sub-dialog differs from the ATTRIBUTE clause of the standalone INPUT statement. The ATTRIBUTE clause has the following syntax:



Element	Description
BOOL Expression	A 4GL expression that returns either 1 (meaning TRUE) or 0 (meaning FALSE) or literal 1 or 0
Integer Expression	A 4GL expression that returns an integer value or a literal integer value that specifies the article help index.
Character Expression	A literal character string or a 4GL expression returning a character string.



Note: This ATTRIBUTE clause does not include any colour or intensity attributes, since it is advised that you use the themes to adjust the appearance of your 4GL programs rather than use 4GL attributes.

The ATTRIBUTE clause of the INPUT sub-dialog can include only up to three attributes and is a reduced version of the [ATTRIBUTE clause](#) of the INPUT statement. The ACCEPT, CANCEL, FIELD ORDER FORM and UNBUFFERED control attributes supported by the standalone INPUT statement cannot be used in the INPUT sub-dialog ATTRIBUTE Clause. Their usage will result in a compilation error.

The attributes supported by the INPUT sub-dialog have the following effect:

- The WITHOUT DEFAULTS attribute accepts values TRUE or FALSE and behaves in the same way as the [WITHOUT DEFAULTS](#) keywords in a standalone INPUT statement. The TRUE value results in the input being done without defaults, whereas FALSE causes the default values to be shown in the



fields. If this attribute is omitted, the default values are displayed. See WITHOUT DEFAULTS attribute of the INPUT statement for the details.

- The HELP attribute functions in the same way as the HELP clause of a standalone INPUT statement. See HELP attribute of the INPUT statement for more details.
- The NAME attribute specifies a unique name for the INPUT clause which can then be used in the form for setting dialog-specific actions. For example, if an INPUT clause has the name "my_input", then you can specify a widget on the form with action "my_input.my_action" which will be used exclusively during the processing of the "my_input" INPUT clause. For more information see "Lycia Developers Guide". See NAME attribute of the INPUT statement.

Here is an example of the INPUT clause with attributes:

```
DIALOG
    INPUT a,b FROM f001, f002 ATTRIBUTE(HELP=5, NAME="my_input")
    END INPUT
    INPUT BY NAME c, d ATTRIBUTE(WITHOUT DEFAULTS=1, NAME="s_input")
    END INPUT
    ...
END DIALOG
```

The INPUT Control Clause

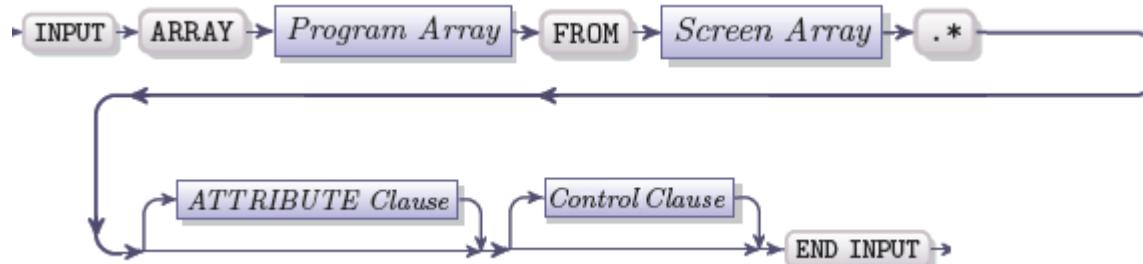
The control clauses have almost the same syntax and behaviour as the [control clauses](#) of the standalone INPUT statement with a single exception. The ON IDLE control clause cannot be used in the INPUT clause of the DIALOG statement, since its role is taken by the [ON IDLE block](#) of the DIALOG statement itself which is described later in this chapter.

The END INPUT Keywords

The END INPUT keywords are obligatory for every INPUT clause of the DIALOG statement, even if the INPUT clause does not contain any control blocks.

The INPUT ARRAY Clause

The INPUT ARRAY Clause is similar in syntax and behaviour to the standalone INPUT ARRAY statement, though there are some differences. For the general description of the INPUT ARRAY behaviour and purpose refer to the [INPUT ARRAY](#) chapter.



Element	Description
Program Array	The name of a variable of the static or dynamic ARRAY of RECORD data type.

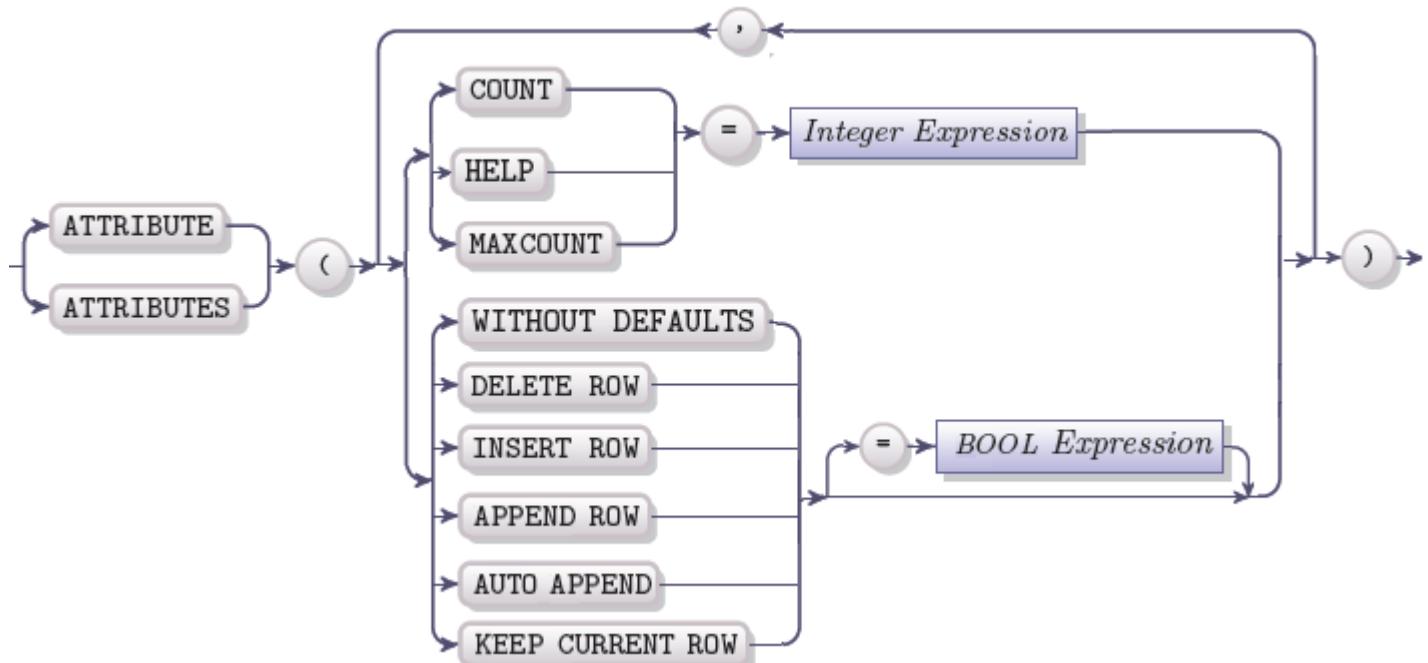


Screen Array	The name of a screen array declared in the form file.
ATTRIBUTE Clause	The list of attributes of the INPUT ARRAY clause. It is a reduced version of the attributes of the standalone INPUT ARRAY statement.
Control Clause	INPUT ARRAY control clauses.

The INPUT ARRAY clause does not have the WITHOUT DEFAULTS keywords or the HELP clause and has a reduced ATTRIBUTE clause.

The ATTRIBUTE Clause

The ATTRIBUTE clause differs much from the ATTRIBUTE clause if the standalone INPUT ARRAY statement. The ATTRIBUTE clause has the following syntax:



Element	Description
BOOL Expression	A 4GL expression that returns either 1 (meaning TRUE) or 0 (meaning FALSE) or literal 1 or 0
Integer Expression	A 4GL expression that returns an integer value or a literal integer value.

	Note: This ATTRIBUTE clause does not include any colour or intensity attributes, since it is advised that you use the themes to adjust the appearance of your 4GL programs rather than use 4GL attributes.
--	---

The ATTRIBUTE block of the INPUT ARRAY clause is a reduced version of the ATTRIBUTE clause of the INPUT ARRAY statement. The reduced clause does not include the colour and intensity attributes. It also forbids the ACCEPT, CANCEL, FIELD ORDER FORM and UNBUFFERED control attributes usage, since these functions are transferred to the DIALOG statement.

It has the following attributes:



- COUNT - defines the number of rows in a program array. See [COUNT](#) attribute of the INPUT ARRAY statement.
- MAXCOUNT - Defines the maximum number of data rows that can be entered into the program array. See [MAXCOUNT](#) attribute of the INPUT ARRAY statement.
- HELP - specifies the help article index to be used when the cursor is in the INPUT ARRAY fields. See [HELP](#) attribute of the INPUT ARRAY statement.
- WITHOUT DEFAULTS - if present or if present and set to 1 the fields display the values of the underlying variables rather than the default field values. See [WITHOUT DEFAULTS](#) attribute of the INPUT ARRAY statement.
- DELETE ROW - if present or if present and set to 1 allows the user to delete program array rows with the delete key. See [DELETE ROW](#) attribute of the INPUT ARRAY statement.
- INSERT ROW - if present or if present and set to 1 allows the user to insert program array rows with the insert key. See [INSERT ROW](#) attribute of the INPUT ARRAY statement.
- APPEND ROW - if present or if present and set to 1 allows the user to append program array rows at the end of the program array. See [APPEND ROW](#) attribute of the INPUT ARRAY statement.
- AUTO APPEND - if present or if present and set to 1 specifies that temporary rows will be created at the end of the program array, when needed. See [AUTO APPEND](#) attribute of the INPUT ARRAY statement.
- KEEP CURRENT ROW - if present or if present and set to 1 allows specifies that the current row will remain highlighted after the execution of the given instruction. See [KEEP CURRENT ROW](#) attribute of the INPUT ARRAY statement.

The INPUT ARRAY Control Clause

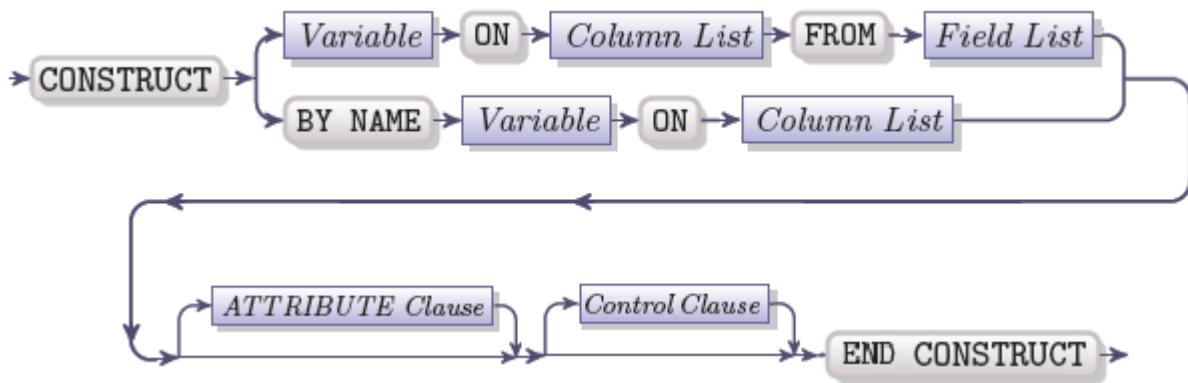
These control clauses have almost the same syntax and behaviour as the [control blocks](#) of the standalone INPUT ARRAY statement with a single exception. The ON IDLE control block cannot be used in the INPUT ARRAY clause of the DIALOG statement, since its role is taken over by the [ON IDLE block](#) of the DIALOG statement itself which is described later in this chapter.

The END INPUT Keywords

The END INPUT keywords are obligatory for every INPUT ARRAY clause of the DIALOG statement, even if the INPUT ARRAY clause does not contain any control blocks.

The CONSTRUCT Clause

The CONSTRUCT Clause is similar in syntax and behaviour to the standalone CONSTRUCT statement, though there are some differences. For the general description of the CONSTRUCT behaviour and purpose refer to the [CONSTRUCT](#) chapter.

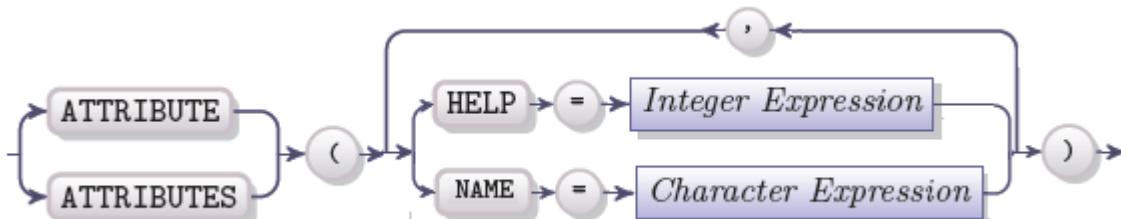


Element	Description
Variable	The name of a variable of the static or dynamic ARRAY or RECORD data type.
Column List	The names of database columns separated by commas.
Field List	The list of form fields separated by commas used to enter the query parameters.
ATTRIBUTE Clause	The list of attributes of the CONSTRUCT clause. It is a reduced variant of the attributes of the standalone CONSTRUCT statement.
Control Clause	CONSTRUCT control clause.

The main difference between the CONSTRUCT clause and the stand alone CONSTRUCT statement is that the clause does not have the HELP block and its ATTRIBUTE clause can contain fewer attributes.

The ATTRIBUTE Clause

The ATTRIBUTE clause of the CONSTRUCT block can have the following attributes:



Element	Description
Character Expression	A 4GL expression returning a character string or a literal character string.
Integer Expression	A 4GL expression that returns an integer value or a literal integer value that specifies the article help index.

- The HELP attribute functions in the same way as the HELP clause of a standalone CONSTRUCT statement. See HELP attribute of the CONSTRUCT statement.
- The NAME attribute specifies a unique name for the CONSTRUCT clause which can then be used in the form for setting dialog-specific actions. See NAME attribute of the CONSTRUCT statement.



The CONSTRUCT sub-dialog does not support the ACCEPT, CANCEL, FIELD ORDER FORM and UNBUFFERED control attributes in contrast to the standalone CONSTRUCT statement. Their usage will result in the compilation error.

The CONSTRUCT Control Clauses

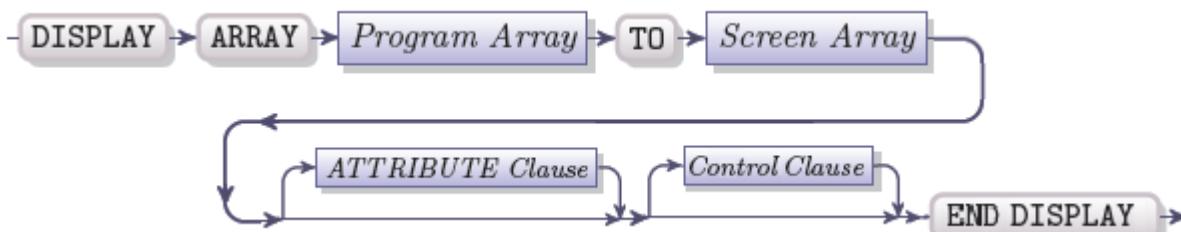
The CONSTRUCT clause has practically the same [control blocks](#) as the CONSTRUCT statement. The only clause which cannot be used in the CONSTRUCT clause is the [ON IDLE block](#), since this functionality was transferred to the DIALOG statement.

The END CONSTRUCT Keywords

The END INPUT keywords are obligatory for every CONSTRUCT clause of the DIALOG statement, even if the CONSTRUCT clause does not contain any control blocks.

The DISPLAY ARRAY Clause

The DISPLAY ARRAY Clause is similar in syntax and behaviour to the standalone DISPLAY ARRAY statement, though there are some differences. For the general description of the DISPLAY ARRAY behaviour and purpose refer to the [DISPLAY ARRAY](#) chapter.

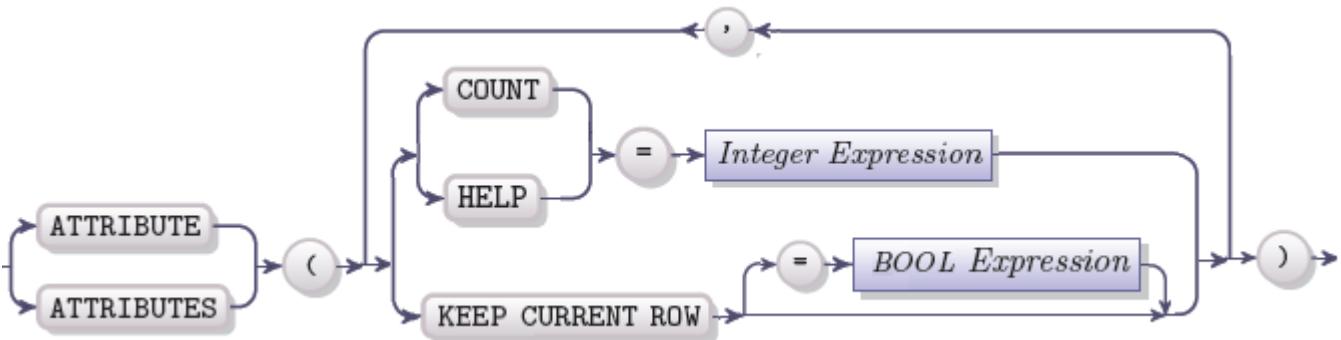


Element	Description
Program Array	The name of a variable of an array of record data type.
Screen Array	The name of the screen array where the program array should be displayed.
ATTRIBUTE Clause	The list of attributes of the DISPLAY ARRAY clause. It is a reduced variant of the attributes of the standalone DISPLAY ARRAY statement.
Control Clause	DISPLAY ARRAY control clause.

The DISPLAY ARRAY clause does not have the HELP clause and has fewer attributes if compared to the DISPLAY ARRAY statement.

The ATTRIBUTE Clause

The ATTRIBUTE clause of the DISPLAY ARRAY block does not include colour or intensity attributes and can contain only up to three attributes:



Element	Description
BOOL Expression	A 4GL expression that returns either 1 (meaning TRUE) or 0 (meaning FALSE) or literal 1 or 0
Integer Expression	A 4GL expression that returns an integer value or a literal integer value.

- COUNT defines the number of populated rows in the array. For details see the COUNT attribute of the DISPLAY ARRAY statement.
- HELP attribute specifies the entry in the help file. For details see the HELP attribute of the DISPLAY ARRAY statement.
- KEEP CURRENT ROW attribute keeps the current row highlighted after the execution of the instructions. For details see the KEEP CURRENT ROW attribute of the DISPLAY ARRAY statement.

In contrast to the standalone DISPLAY ARRAY statement the DISPLAY ARRAY sub-dialog does not support the ACCEPT, CANCEL, FIELD ORDER FORM and UNBUFFERED control attributes. Their usage always results in the compilation error.

The DISPLAY ARRAY Control Clauses

The DISPLAY ARRAY clause has practically the same [control blocks](#) as the DISPLAY ARRAY statement. The only clause which cannot be used in the DISPLAY ARRAY clause is the [ON IDLE block](#), since this functionality was transferred to the DIALOG statement.

The END DISPLAY Keywords

The END DISPLAY keywords are obligatory for every DISPLAY ARRAY clause of the DIALOG statement, even if the DISPLAY ARRAY clause does not contain any control blocks.

The DIALOG Control Clauses

Since each input or display clause within the DIALOG statement can have its own control clauses, the control clauses of the DIALOG statement are aimed at regulating the general execution of the DIALOG instructions. The DIALOG control clauses may appear only after the last sub-dialog clause within the DIALOG statement.

These are the dialog clauses in the order of their execution:

- BEFORE DIALOG
- User activated clauses:
 - ON KEY



- ON ACTION
- COMMAND
- AFTER DIALOG

The Order of Execution

Since the DIALOG statement and its sub-dialogs have many control clauses it is important to bear in mind the order of their execution. The following table lists the possible control clauses and their interaction peculiar to the DIALOG statement. The order of execution of the control clauses within a single dialog is typically the same as the order of execution of these control clauses for the corresponding standalone statement and is described in the chapters covering the corresponding statements.

Event	Order of Execution
The dialog begins	<ol style="list-style-type: none">1. BEFORE DIALOG2. BEFORE INPUT/BEFORE CONSTRUCT/BEFORE DISPLAY of the first sub-dialog in the list.3. BEFORE ROW if the first sub-dialog is DISPLAY ARRAY or INPUT ARRAY4. BEFORE FIELD if the first sub-dialog is not DISPLAY ARRAY.
Cursor enters the next sub-dialog (for the INPUT or CONSTRUCT sub-dialogs)	<ol style="list-style-type: none">1. The control clauses of the previous sub-dialog.2. The BEFORE INPUT/BEFORE CONSTRUCT clause.3. The BEFORE FIELD clause for the first field
Cursor enters the next sub-dialog (for the DISPLAY ARRAY sub-dialog)	<ol style="list-style-type: none">1. The control clauses of the previous sub-dialog.2. BEFORE DISPLAY clause3. BEFORE ROW clause
Cursor enters the next sub-dialog (for the INPUT ARRAY sub-dialog)	<ol style="list-style-type: none">1. The control clauses of the previous sub-dialog2. BEFORE INPUT clause3. BEFORE ROW clause4. BEFORE FIELD clause for the first field in the array
Cursor leaves the current sub-dialog (for the INPUT or CONSTRUCT sub-dialogs)	<ol style="list-style-type: none">1. ON CHANGE clause if the value in the current field was changed.2. AFTER FIELD of the current field.



3. AFTER INPUT/AFTER CONSTRUCT for the current sub-dialog.

4. The control clauses of the next sub-dialog you enter.

Cursor **leaves** the current sub-dialog (for the DISPLAY ARRAY sub-dialog)

1. AFTER ROW clause.

2. AFTER DISPLAY clause.

3. The control clauses of the next sub-dialog.

Cursor **leaves** the current sub-dialog (for the INPUT ARRAY sub-dialog)

1. ON CHANGE clause, if the value in the current field was changed.

2. AFTER FIELD for the current field.

3. AFTER INSERT if the current row was just added or ON ROW CHANGE if a value in the current row was changed.

4. AFTER ROW clause.

5. AFTER INPUT clause

6. The control clauses of the next sub-dialog you enter.

Cursor moves from field x to field y, if the fields are located in the same sub-dialog

1. ON CHANGE x, if the field value was changed.

2. AFTER FIELD x clause.

3. BEFORE FIELD y clause.

Cursor moves from field x to field y, if the fields are located in different sub-dialogs

1. ON CHANGE x clause, if the field was changed.

2. AFTER FIELD x.

3. AFTER INPUT/AFTER CONSTRUCT clause of the current sub-dialog.

4. BEFORE INPUT/BEFORE CONSTRUCT clause of the next sub-dialog.

5. BEFORE FIELD y.

The ACCEPT DIALOG keywords are executed

1. ON CHANGE clause if the value in the current field was changed.

2. AFTER FIELD for the current field (if the current sub-dialog is not DISPLAY ARRAY).

3. AFTER INSERT (if the current sub-dialog is INPUT ARRAY if the current row was just added) or ON ROW CHANGE (if the current sub-dialog is INPUT ARRAY if the values in the current row



changed).

4. AFTER ROW (if the current sub-dialog is INPUT ARRAY or DISPLAY ARRAY).
5. AFTER INPUT/AFTER CONSTRUCT/AFTER DISPLAY clause of the current sub-dialog.
6. AFTER DIALOG clause.

The EXIT DIALOG keywords are executed	None of the control clauses are executed, the program focus moves to the first statement following the END DIALOG keywords.
---------------------------------------	---

The CONTINUE DIALOG statement is executed

1. BEFORE INPUT/BEFORE CONSTRUCT/BEFORE DISPLAY of the first sub-dialog in the list.
2. BEFORE ROW if the first sub-dialog is DISPLAY ARRAY or INPUT ARRAY.
3. BEFORE FIELD if the first sub-dialog is not DISPLAY ARRAY.

The BEFORE DIALOG Clause

The BEFORE DIALOG keywords are optional. The instructions in this clause are executed before the form is made active and the fields become available for input and display. This clause consists of the BEFORE DIALOG keywords and any subsequent statements between the BEFORE DIALOG clause and another control clause or the END DIALOG keywords, if there are no additional control clauses.

```
DIALOG ATTRIBUTE (UNBUFFERED, FIELD FORM ORDER)
    INPUT BY NAME my_rec.* ATTRIBUTE (HELP=5)
    END INPUT
    ...
    BEFORE DIALOG
        DISPLAY "Enter the details"
    END DIALOG
```

The ON IDLE Clause

The ON IDLE clause specifies the actions the application should undertake in case it becomes inactive for the specified time. The syntax of this block is as follows:

```
ON IDLE seconds_counter
```

Here, seconds_counter is either a literal integer or a 4GL expression returning an integer that specifies the number of seconds the application must be idle to trigger the execution of this clause.

It is advisable to detect long periods of inactivity (e.g. 30 seconds or more) to return the control to the program in case if there is no reaction from the user, for example the user can be away from the keyboard.



In this case the ON IDLE block will be triggered and some code can be executed, before the control is returned to the user.

```

DIALOG

    INPUT ARRAY my_arr FROM scr_arr.*

    END INPUT

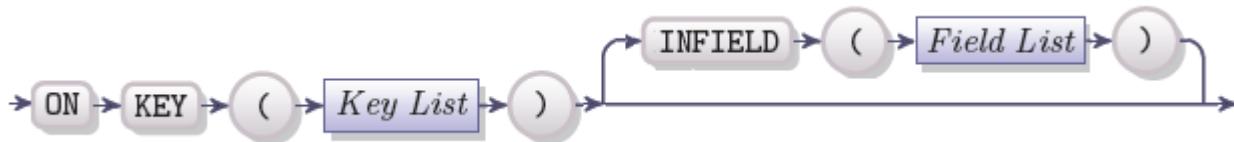
    ...

    ON IDLE 60
        LET info = fgl_winquestion("Timeout Warning",
            "Do you want to continue?", "Yes", "Yes|No",
            "info", 1)
        IF info = "Yes" THEN
            CONTINUE DIALOG
        ELSE
            EXIT DIALOG
        END IF
    END DIALOG

```

If the seconds counter value evaluates to zero or to a negative integer, the ON IDLE clause is ignored and the timeout is disabled. If the seconds counter is represented by a variable, it is evaluated only once during the initialization of the dialog and any changes of its value during the execution of the dialog will have no effect.

The ON KEY Clause



Element	Description
Field List	A list of one or more form fields separated by commas
Key List	A list that consists of one to four key names separated by commas

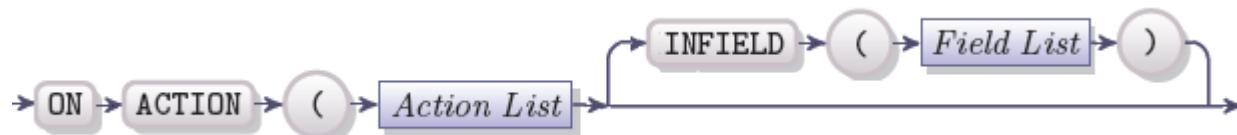
The ON KEY clauses can be included into the DIALOG statement to specify the behaviour of the keys pressed while the input is performed. The keys can be assigned to form widgets and the actions will be triggered when such widget is pressed. However, if a key combination is not assigned to any widget, it can be activated with the help of the keyboard. One DIALOG statement can contain any number of the ON KEY clauses, they can be placed in any order.

The ON KEY clause of the DIALOG statement can also include an optional infield() operator. Even though the DIALOG does not reference fields directly, any fields that are included into its input and display clauses can be used in this instruction. This operator specifies a field or a list of fields for which the ON KEY clause will be triggered, if the key referenced by it is pressed. Only if the cursor is located in one of the listed keys during the key press, the clause will be triggered.



The DIALOG statement ON KEY clause functions in the same way as the INPUT statement ON KEY clause. Thus for more details on the ON KEY clause as well as for the information about the allowed key names see the [ON KEY clause](#) of the INPUT statement.

The ON ACTION Clause



Element	Description
Field List	A list of one or more form fields separated by commas
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks

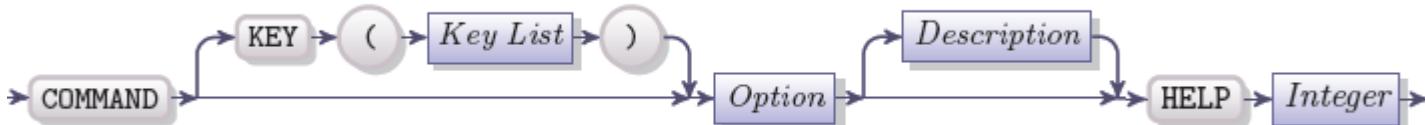
The actions which you have assigned to the widgets in the form specification file can be used in an ON ACTION clause. The ON ACTION clauses are treated by 4GL very much like the ON KEY clauses. 4GL executes the statements specified in it, when the user presses a button or other widget on the form to which the corresponding action is assigned. Whereas an ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard, an ON ACTION clause is activated with the help of a form widget. The action list can contain several actions separated by commas.

The ON ACTION clause of the DIALOG statement can also include an optional infield() operator. Even though the DIALOG does not reference fields directly, any fields that are included into its input and display clauses can be used in this instruction. This operator specifies a field or a list of fields for which the ON ACTION clause will be triggered, if the action referenced by it is activated. Only if the cursor is located in one of the listed keys during the key press, the clause will be triggered.

The ON ACTION clause of the DIALOG statement functions in the same way as that of the INPUT statement. Thus for more details see the INPUT statement [ON ACTION clause](#) description.

The COMMAND Clause

The COMMAND clause resembles the COMMAND clause of the MENU statement.



Element	Description
Key List	A list of one or more key names separated by commas.
Option	The name of the option.
Description	The description of the option.
Integer	A literal integer or an integer expression that specifies the help entry index.



The AFTER DIALOG Clause

The AFTER DIALOG clause is executed before the DIALOG statement is completed. A dialog can be completed either when the cursor moved out from the last field included into the dialog, or when the ACCEPT DIALOG keywords are executed. The AFTER DIALOG clause is not executed, if the dialog is terminated by means of the EXIT DIALOG keywords.

If the AFTER DIALOG statement includes the NEXT FIELD or CONTINUE DIALOG keywords, the DIALOG statement will continue its execution, so it is advisable to add these instructions only in the form of conditional statements.

```
DIALOG
...
AFTER DIALOG
  IF cuts_name IS NULL THEN
    NEXT FIELD cust_name
  END IF
END DIALOG
```

Keywords Used in DIALOG Control Clauses

There is a number of specific keywords that can be used in the DIALOG control clauses, they are:

- ACCEPT DIALOG
- CONTINUE DIALOG
- EXIT DIALOG
- NEXT FIELD

All the above listed keywords with the exception of NEXT FIELD keywords can only be used within the control clauses of the DIALOG statement.

The ACCEPT DIALOG Keywords

These keywords are necessary to exit a dialog. Since the dialog is a container for other input statements, the default functioning of the Accept key was disabled in order to keep all of the included dialogs active and prevent termination of some of them prematurely. Thus pressing the Accept key will have no effect by default and will end neither the dialog itself nor the sub-dialog clauses.

To finish the dialog you need to explicitly include the ACCEPT DIALOG keywords in one of the DIALOG control blocks. When 4GL encounters these keywords, it writes the entered values into the corresponding underlying variables, executes the AFTER FIELD and AFTER ROW instructions, if they are present for the current field or row, then the AFTER INPUT/AFTER DISPLAY/AFTER CONSTRUCT clause of the current sub-dialog, if any, and then the AFTER DIALOG clause. After that it moves the first statement following the END DIALOG keywords, unless the AFTER DIALOG clause included the NEXT FEILD or CONTINUE DIALOG statement.



	Note: The ACCEPT DIALOG keywords are necessary to complete the dialog statement.
---	---

```
DIALOG
...
ON KEY ACCEPT
    ACCEPT DIALOG
END DIALOG
```

The EXIT DIALOG Keywords

The EXIT DIALOG keywords along with the ACCEPT DIALOG keywords are required to terminate the DIALOG statement. The dialog cannot be terminated by pressing the Interrupt key which default functionality is disabled for this statement. Pressing the Interrupt key will have no effect by default and will not terminate either the dialog or any of the sub dialogs.

To exit the DIALOG you must explicitly specify the EXIT DIALOG keywords in one of the dialog control clauses. When the EXIT DIALOG key words are executed, the program execution resumes at the first statement following the END DIALOG keywords. The AFTER DIALOG clause, if it is present, is not executed in this case. The corresponding AFTER FIELD, AFTER ROW AND AFTER INPUT/AFTER CONSTRUCT/AFTER DISPLAY clauses are also skipped.

	Note: The ACCEPT DIALOG keywords are necessary to exit the dialog statement.
---	---

```
DIALOG
...
ON KEY ACCEPT
    ACCEPT DIALOG
    ON KEY (F12)
        EXIT DIALOG
END DIALOG
```

The CONTINUE DIALOG Keywords

The CONTINUE DIALOG keywords resume the execution of the DIALOG statement. Any other statements below these keywords in the control clause where it is located are ignored and the control is returned to the user. This instruction may be useful in case of deeply nested conditional and other statements in the DIALOG control blocks. If the CONTINUE DIALOG statement is placed in the AFTER DIALOG clause, you should use it in a conditional statement, otherwise the dialog will not be possible to exit by means of the ACCEPT DIALOG instruction.



When the execution of the dialog resumes, the BEFORE INPUT/BEFORE CONSTRUCT/BEFORE DISPLAY clause of the first sub dialog is executed. Then the BEFORE FIELD / BEFORE ROW clause is executed for the first field or row of this sub-dialog, if they are present.

The NEXT FIELD Keywords

The NEXT FIELD keywords can be used on the DIALOG statement even though strictly speaking the DIALOG does not reference any fields directly and the field are manipulated by the nested input and display clauses. Even though the DIALOG does not reference fields directly, any fields that are included into its input and display clauses can be used in this instruction.

The NEXT FIELD keywords can be followed by:

- NEXT keyword, which will move the cursor to the next field:

```
NEXT FIELD NEXT
```

- PREVIOUS keyword, which will move the cursor to the previous field:

```
NEXT FIELD PREVIOUS
```

- The name of the field to which the cursor will be moved:

```
NEXT FIELD field-name
```

The NEXT FIELD keywords move the cursor to the specified field. If the current field and the next field are located within the same sub-dialog (e.g. within the same INPUT clause), the AFTER FIELD or AFTER ROW clauses for the current field are not executed, but the BEFORE FEILD and BEFORE ROW for the next field into which the cursor is moved are executed. If the current field and the next field are located within different sub-dialogs (i.e. in different INPUT clauses), the corresponding AFTER FIELD, AFTER ROW, AFTER INPUT/AFTER DIAPLAY/AFTER CONSTRUCT are executed for the current field and then BEFORE INPUT/BEFORE CONSTRUCT/BEFORE DISPLAY, BEFORE ROW and BEFORE FIELD will be executed for the field into which the cursor was moved.

With the NEXT and PREVIOUS keywords the order of fields is important. If the FORM FIELD ORDER attribute is set for the current DIALOG statement, the cursor will be moved in accordance with the TABINDEX form attribute. If this attribute is absent, the order of fields corresponds to their order of appearance in the corresponding sub-dialog declaration. If the NEXT FIELD NEXT keywords are used for the last field of the current sub-dialog, the cursor moves to the first field of the next sub-dialog. If the NEXT FIELD PREVIOUS keywords are used for the first field of the current sub-dialog, the cursor will be moved to the last field of the previous sub-dialog.

The ui.Dialog Class

Lycia also has a built-in class use to manipulate the DIALOG instructions. This class offers a number of methods aimed at manipulating dialog properties dynamically. Inside the dialog statement, the predefined keyword DIALOG represents the current dialog object. The methods can be executed using this object. The DIALOG keyword can be used together with the methods only within the DIALOG statement and cannot be used outside this statement. The DIALOG object is local to the DIALOG statement, but can be passed to functions in case of necessity.



The DIALOG class can be used in the following way in the DIALOG control clauses:

```
ON CHANGE cust_country
CALL DIALOG.setFieldActive("cust_address", (cust_country IS NOT NULL))
```

For the detailed information on the ui.Dialog class see "Lycia Built-In Functions" Guide.

Example

There is an example where a DIAPLAY ARRAY and an INPUT are executed simultaneously in a DIALOG statement:

```
MAIN
DEFINE my_rec RECORD
    fname, lname STRING,
    age INT
END RECORD,
arr DYNAMIC ARRAY OF RECORD
    fname, lname STRING,
    age INT
END RECORD,
i INT

LET arr[1].fname="John"
LET arr[1].lname="Smith"
LET arr[1].age=32
LET arr[2].fname="Anna"
LET arr[2].lname="Thomson"
LET arr[2].age=29
LET i = 2

LET my_rec.fname="Enter First Name"
LET my_rec.lname="Enter Family Name"
LET my_rec.age=0

OPEN FORM dialog_form FROM "dial_f"
DISPLAY FORM dialog_form

DIALOG ATTRIBUTES (UNBUFFERED, FIELD ORDER FORM)

INPUT BY NAME my_rec.* ATTRIBUTE (WITHOUT DEFAULTS = 1)
```



```
END INPUT

DISPLAY ARRAY arr TO scr./*
BEFORE DISPLAY
CALL fgl_message_box("Scroll down to view the array.")
END DISPLAY

BEFORE DIALOG
CALL set_count(i)
ON KEY(Accept)
ACCEPT DIALOG
ON KEY(F12)
EXIT DIALOG
AFTER DIALOG
IF fgl_winquestion("Record Added",
"Do you want to add another record?",
"Yes", "Yes|No", "question", 1) = "Yes" THEN
LET i=i+1
LET arr[i].fname=my_rec.fname
LET arr[i].lname=my_rec.lname
LET arr[i].age=my_rec.age
INITIALIZE my_rec.* TO NULL
CONTINUE DIALOG
END IF
END DIALOG

END MAIN
```

When used with the following form file, this program will allow to add the records to the list and display them without stopping the input:

```
DATABASE formonly

SCREEN {

    [fname      ] [lname      ] [age      ]
    [a_fname    | a_lname    | a_age    ]
```



```
[a_fname | a_lname | a_age ]  
}  
  
}
```

ATTRIBUTES

```
fname=formonly.fname;  
lname=formonly.lname;  
age=formonly.age;  
a_fname=formonly.a_fname;  
a_lname=formonly.a_lname;  
a_age=formonly.a_age;
```

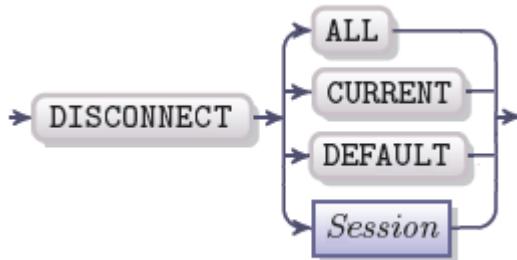
INSTRUCTIONS

```
DELIMITERS "||"  
SCREEN GRID scr[5] (a_fname, a_lname, a_age)
```



DISCONNECT

The DISCONNECT statement is used to close a database connection session opened previously by the [CONNECT TO](#) statement. It closes the given database connection, here is its syntax:



Element	Description
Session	A string expression indentifying the session name of the database connection to be terminated.

This statement can close the database connections created both by means of the CONNECT TO statement and DATABASE statement. If a DISCONNECT statement is executed while a transaction is active, the transaction is automatically rolled back. Once a connection was disconnected, you cannot use the SET CONNECTION statement to specify it as the current connection. To use the disconnected session again you need to reopen it using the CONNECT TO statement.

	Note: No other session becomes active automatically once the current session gets disconnected. To be able to work with the SQL statements further, you need to set a new current connection either by executing a new CONNECT TO statement or by setting the current connection using the SET CONNECTION statement.
--	---

DISCONNECT ALL

The ALL keyword following the DISCONET statement terminates all connections to all databases previously opened, even the default connection opened by the DATABASE statement. If you execute this statement, you will not be able to execute SQL statements, even though you may have the DATABASE statement at the beginning of the main module defining the default database. To be able to execute SQL statements again, the program must execute either DATABASE or CONNECT TO statement after the DISCONNECT ALL statement was processed.

DISCONNECT CURRENT

The CURRENT keyword following the DISCONNECT statement terminates the current database connection. The current connection is the session of the last executed CONNECT TO statement or the session that was made the current one by the SET COONNECTION statement . If there were no CONNECT TO statements prior to it, the session established by the DATABASE statement is considered to be current.



Disconnection a Named Session

You can disconnect any session that was previously opened regardless of it being the current session or not. The DISCONNECT statement must be followed by the name of the session to be disconnected. The name of the session is case sensitive, it can be one of the following:

- The name of the session specified in the AS clause of the CONNECT TO statement.
- The database name specified in the CONNECT TO statement, if the AS clause is omitted.
- The DEFAULT keyword used instead of the session name disconnects the default database session established by the DATABASE statement.

The example below leaves only one database session running, which is the session named "db1":

```
DATABASE db

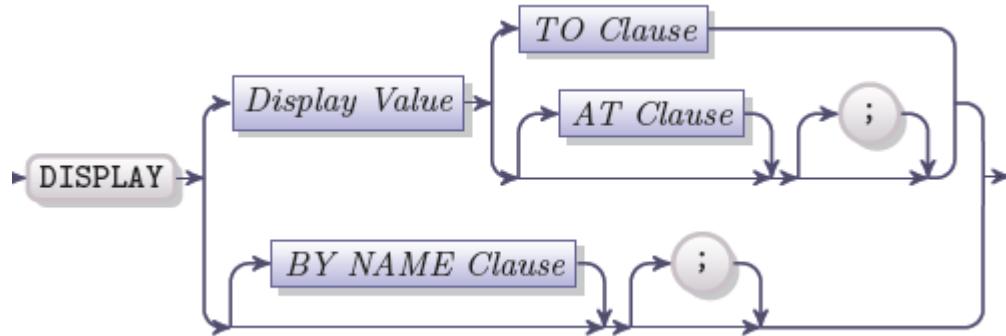
MAIN
...
CONNECT TO "db1"
CONNECT TO "db2"
CONNETC TO "db3" AS "session3"
...
DISCONNECT "session3"
DISCONNECT "db2"
DISCONNECT DEFAULT
SET CONNECTION "db1"

END MAIN
```



DISPLAY

The DISPLAY statement is used to display information on the screen. Different clauses that follow the DISPLAY statement determine where on the screen the information will be displayed.



Element	Description
Display Value	An actual value enclosed in the double quotation marks, a variable which value is to be displayed or their combination separated by commas.
TO Clause	An optional clause that specifies the form fields to which the Display value will be displayed
AT Clause	An optional clause that specifies the coordinates in the 4GL screen or window where the Display value will be displayed
BY NAME Clause	An optional clause that implicitly associates variables with the form fields in which their values will be displayed

The information will be displayed to the place defined by the optional display clauses. If no optional clause is present, the output will be performed to 4GL screen or to the Output console.

The Output to the Screen or Window

When a value is displayed to the 4GL screen or a 4GL window it either can contain the AT clause which specifies the exact position of the displayed information or it may be without the AT clause.

The DISPLAY Statement without the AT Clause

The DISPLAY statement can be followed by the Display Value and not followed by any optional display clauses:



The output will be performed differently in Character mode and in GUI mode:

- in Character mode: to the current screen at the default position which is the top left corner of the 4GL screen
- in GUI mode: to the Output console



The DISPLAY statement without the AT clause cannot be followed by an ATTRIBUTE clause, whereas a DISPLAY statement with any of the display optional clauses can.

The value can be displayed with the left offset, if the keyword COLUMN is used. The COLUMN keyword defines the offset by which the displayed value will be shifted to the right. The character string in the example below will be displayed at the fifteenth column of either the screen or Output console. If the length of the "var1" value is longer than 15 characters, the character string will be shifted to the right and displayed next to it.

```
DISPLAY var1, COLUMN 15, "definition"
```

You can use the SPACE operator to insert whitespaces between the display values. The integer preceding the SPACE operator specifies the number of white spaces inserted between the last character of the first value and the first character of the second one:

```
DISPLAY "first_value", 5 SPACE, "second_value"
```

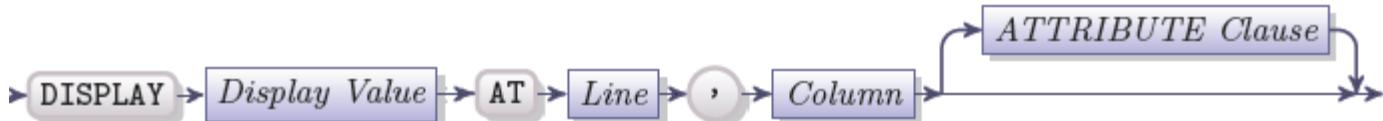
In this case, the position of the second value does not have the absolute coordinates, it has coordinate relative to the position of the previous display value.



Note: The Output console can be prevented from being displayed with the help of script options. It is usually used at the development stage and is disabled in the release version of a program

The DISPLAY ... AT Statement

The DISPLAY ... AT statement has the following structure:



Element	Description
Display Value	An actual value enclosed in the double quotation marks, a variable which value is to be displayed or their combination separated by commas.
Line	A 4GL expression returning a positive integer that specifies a line of the current 4GL window or 4GL screen.
Column	A 4GL expression returning a positive integer that specifies a column on the specified line of the current 4GL window or 4GL screen.
ATTRIBUITE Clause	An optional ATTRIBUITE clause which affects the display attributes applied to the Display Value

Line is the line of the current window or 4GL screen, the *column* is the left offset that specifies the horizontal coordinates of the output with regard to the left side of a window or screen. The *line* and the *column* can be



either literal integers, or integer expressions (i.e. variables of INT or SMALLINT data types). Both the *line* and the *column* should be specified.

```
#the value is displayed at the 2nd column of the 5th line
DISPLAY var1 AT 5,2
```

The output from the DISPLAY ... AT statement will be performed at the specified line and column of the current window or of the 4GL screen, if it is used as the current window.



Note: The values specified as the coordinates are not converted, thus you cannot use character values in the expressions, they will not be converted to integer values and you will receive an error. Only INT and SMALLINT variables can be used in the expressions.

The Display Format

The Display Value of the DISPLAY statement can contain several literal values or variables. They should be separated by commas.

```
DISPLAY variable1, "value", variable2 AT line_var, 3
```

You can also use the following tools to format the output:

- The USING keyword changes the display format of variables of the time data types
- The CLIPPED keyword truncates trailing whitespaces from the right
- The *integer* SPACE operator inserts the specified number of whitespaces into the display string.
- The COLUMN *integer* operator causes the next value to be displayed with the specified left offset



Note: The items, displayed with the help of the same DISPLAY statement are not separated by white spaces automatically, use SPACE or COLUMN operator to insert white spaces where necessary.

Here is an example of the DISPLAY statement used with several operators:

```
DISPLAY "If c=", c CLIPPED, 1 SPACE, "then the expression is correct."
```

The length of the displayed variable depends on its declared data type, if you do not use the CLIPPED operator:

- CHAR – the length displayed is the same as the length declared (blank spaces are included to fit the declared size)
- DATE – 10 characters
- DATETIME – depends on the declared DATETIME qualifier (from 2 to 25)
- DECIMAL – 2+p characters, where p is the declared precision
- FLOAT – 14 characters
- INTEGER - 11 characters
- INTERVAL - depends on the declared INTERVAL qualifier (from 2 to 25)
- MONEY – 3+p characters, where p is the declared precision



- SMALLFLOAT – 14 characters
- SMALLINT – 6 characters
- VARCHAR – the length displayed is the same as the length of the assigned value, no whitespaces added automatically and no user-entered whitespaces are truncated

The AT Clause

AT clause specifies where the information will be displayed in a 4GL window. If the AT clause is omitted, the information will be displayed at the line that follows the line where the cursor currently is in the character mode and in the Output consol in the GUI mode.

You cannot use the AT clause within a DISPLAY statement where the TO or BY NAME keywords are used. The ATTRIBUTE clause cannot be used with a DISPLAY statement that does not contain any other clauses (AT, TO, BY NAME).

The 1,1 coordinates are located in the upper left-hand corner of a current 4GL window or of the 4GL screen. The first number refers to the lines, the larger is the number the lower is the line. The second number refers to the character position; the larger it is, the closer to the right side of the screen or a window the displayed information is situated.

If the coordinates specified in the AT clause exceed the size of a window or a screen, 4GL will not produce an error, but the displayed information will not be visible.

The information displayed remains on the screen until it is overwritten. If the last value to display is the NULL value of the CHAR data type, the whole line to the right from the last displayed value is cleared, if the application is run in character mode. In GUI mode, the line is not cleared and everything displays normal.

4GL reserves the default positions for:

- The MESSAGE line – the second line of the current window
- The ERROR line – The last line of the 4GL screen in the character mode and the last line of the current 4GL window in the graphical client mode
- The PROMPT line – the first line of the current window or the 4GL screen
- The COMMENT line – the last but one line of the 4GL window or screen (it can be shown as a screen tip in GUI clients)

It is advisable that you do not specify the above listed coordinates in the AT clause of a DISPLAY statement, otherwise the corresponding messages will be overwritten. If you need to use these coordinates, you can change the position of the default lines using the OPTIONS statement.

The Output to Form

The information can be displayed in the fields of a screen form with the help of the TO clause or BY NAME keywords. One DISPLAY statement cannot contain the BY NAME keywords and the TO clause at the same time. In both cases you can add an attribute clause to the DISPLAY statement. You cannot use the AT clause when you display the information to a form field.

The values are displayed in a form field depending on their data types:

- Numbers are right-justified, if a number does not fit the field length, an asterisk is put in the field instead (currently 4GL does not display anything)
- Literal strings and TEXT values are left-justified, if a literal string is longer than the field, the superfluous symbols are truncated from the right
- BYTE values cannot be displayed with the help of DISPLAY statement, the field will display <byte value> instead. To display a value of BYTE data type use the PROGRAM attribute



The DISPLAY Statement with the BY NAME Clause

The BY NAME clause is used to display values of variables to form fields when the names of the fields are the same as the names of the variables. Any prefixes (like *formonly*, or table indication) are ignored when the names are matched. The names must be unique and separated by commas.



Element	Description
Display Value	One or more variables separated by commas the values of which are to be displayed.
ATTRIBUTE Clause	An optional ATTRIBUTE clause which affects the display attributes applied to the Display Value

E.g. if you have declared the variables *lname*, *fname*, *address* and *phone* and have the fields in a form with the same names, you can use the BY NAME clause to display the values of these variables to the corresponding fields:

```
DISPLAY BY NAME lname, fname, address, phone
```

The above BY NAME clause displays the values to individual form fields. To display values to a screen record, you may use a program record which members match the members of a screen record in order number and names. The names for the screen record and the program record need not match, but the names of the form fields included into the screen array and the members of the program array must be the same. Use the name of the program record in the DISPLAY statement:

```
DISPLAY BY NAME prog_rec.lname, prog_rec.fname, prog_rec.address,  
prog_rec.phone  
DISPLAY BY NAME prog_rec.lname THRU prog_rec.phone
```

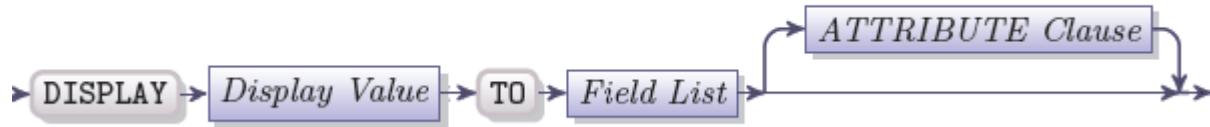
The default screen record is the *formonly* screen record. To display values to a screen array, you must define a screen array. By default, the values are displayed to the first line of an array:

```
DISPLAY BY NAME my_arr[1].lname
```

To display the *lname* value in other elements of an array, you may use the TO clause instead of the BY NAME clause, or use the DISPLAY ARRAY statement.

The DISPLAY Statement with the TO Clause

If the names of the variables and the names of the fields differ, you should use TO clause instead of BY NAME clause. The DISPLAY statement with the TO clause has the following structure:



Element	Description
Display Value	An actual value enclosed in the double quotation marks, a variable which value is to be displayed or their combination separated by commas.
Field List	A list of individual form fields, screen records and their members or screen arrays and their elements
ATTRIBUTE Clause	An optional ATTRIBUTE clause which affects the display attributes applied to the Display Value

You can list the fields individually or you can use screen records or screen arrays, either way they should be separated by commas. You can use the *screen_record.** notation to specify all the fields of a screen record, or you can use the *screen_array[i].** notation to specify all the fields within the *i* line of a screen array. However, the DISPLAY ARRAY statement is more convenient for displaying information to screen arrays.

If an ATTRIBUTE clause is present within the DISPLAY statement, the attributes will be applied to all the fields specified in the TO clause, or implied by the BY NAME clause.

In the example below, all the variables that belong to a program record are displayed to a screen record. The order of the fields in the screen record must be the same as the order of the variables within the program record. They also should correspond to each other in number and their data types should be the same or compatible, but they need not have the same names as in the case with the BY NAME clause.

```
DISPLAY pr_rec.* TO scr_rec.*
```

The DISPLAY ... TO widget Statement

Besides, the DISPLAY statement with the TO clause can also be used to change the appearance and behaviour of a button and a label form objects.

To update the appearance of these widgets, that is image and text label properties setting, or replacing or removing of the already existing ones, you should pass an Uri as an image property value, or a text to be displayed as a widget's text label. The execution algorithm of such a statements is as follows.

Updating the already existing settings:

- If a widget possesses only one of the two properties (image or text), a display value passed to the statement replaces the one, that is set.
- If both properties are already applied to a widget, the image is replaced in the case the display value stands for an Uri. Otherwise, the text label is changed.
- If neither image nor text is set, an image is updated in the case the display value stands for an Uri. Otherwise, the text label is applied.



Removing an image or a text label from a widget:

- Passing the NULL string (a zero length value) as a display value

```
DISPLAY "" TO button_name
```

```
# removes an already existing property settings, if a widget possesses at least one of them;
```

```
# removes a text label if a widget has both an image and text label set (as the string passed as a display value does not stand for an Uri).
```

- Passing the *qx:empty* string as a display value

```
DISPLAY "qx:empty" to button_name
```

```
# always removes only an image property settings
```

As concerns a widget behaviour, it can be modified by passing one of the following symbols '!', '*' or '?' as a display value to the DISPLAY ... TO widget statement.

By default, on a form loading, the dynamic context event state of a widget is activated. For such a behaviour the '?' symbol stands:

```
DISPLAY "?" TO widget_name
```

Symbols '*' and '!', irrespective of the presence of any active events at any moment of a program workflow, stand for a widget enabling and disabling correspondingly. The same result can be obtained using the SetEnable() method: if its value is TRUE (the default behaviour), a widget is active, that means it can be interacted with; otherwise, it becomes disabled (grayed and inaccessible for the user). For more details concerning UI element class, please, refer to the "Querix 4GL UI Types" document.

The ATTRIBUTE clause

The DISPLAY statement can include an ATTRIBUTE clause only if it also contains one of the other optional clauses (AT, TO or BY NAME clause). The ATTRIBUTE clause of the DISPLAY statement has the structure of the common [ATTRIBUTE clause](#). At least one attribute must appear within the parentheses following the ATTRIBUTE clause.

The Numeric, Monetary and Time Values

The display format of monetary and numeric values depends on the MONETARY and NUMERIC categories of the local files. The display format of these values can also be influenced by the DBFORMAT and DBMONEY environment variables and the USING operator.

- The MONEY values can be preceded by a currency symbol, it is set by the DBFORMAT or DBMONEY environment variables and can be changed with the help of the FORMAT attribute



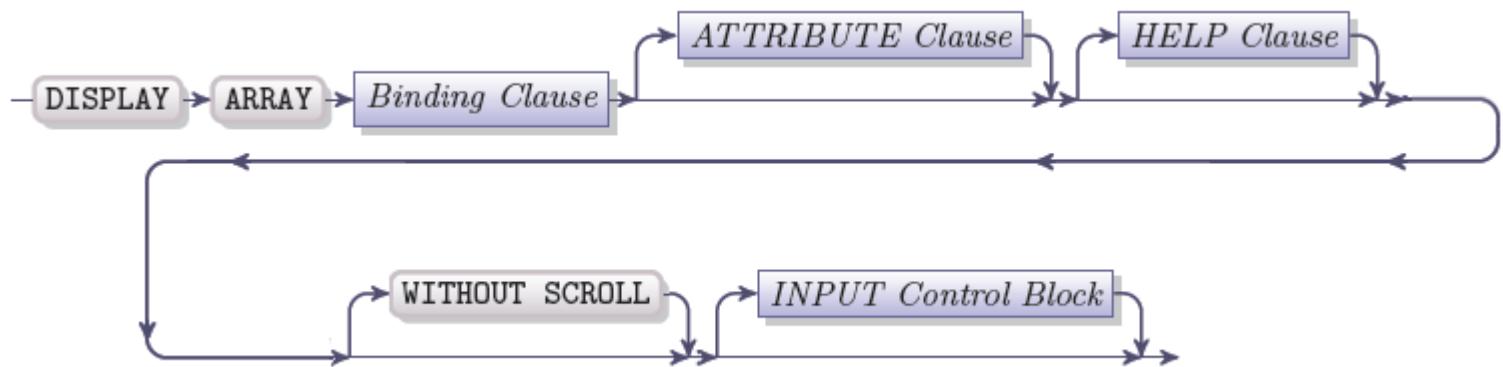
- The separators of thousands are omitted if they are not specified in the FORMAT attribute or in the USING operator
- The decimal separator is displayed for numbers with fractional part by default
- The trailing currency symbols are displayed for the MONEY data type unless otherwise is specified in the FORMAT attribute or USING keyword.
- In some locales (i.e. supporting Italian or Portuguese languages) monetary values are displayed in different formats than other numeric values.

The display format of the time values is affected by the DBDATE, GL_DATE, and GL_DATETIME environment variables and by the USING operator.



DISPLAY ARRAY

The DISPLAY ARRAY statement is a special case of the DISPLAY statement which includes most of the additional clauses common for INPUT and INPUT ARRAY statement as well as its own additional clauses.



Element	Description
Binding Clause	This is an obligatory clause which specifies the program array and the record array to which it must be displayed
ATTRIBUTE Clause	An optional ATTRIBUTE clause where the attributes for the fields used in the DISPLAY ARRAY statement can be specified
HELP Clause	An optional HELP clause where the help number for the DISPLAY ARRAY statement can be specified
WITHOUT SCROLL clause	This optional clause consists of the WITHOUT SCROLL keywords.
INPUT Control Block	An optional DISPLAY control block which controls the display of the data

The DISPLAY ARRAY statement is used to display program arrays to screen arrays; it may include optional clauses.

To use the DISPLAY ARRAY statement, follow these steps:

1. Specify a screen array in a form specification file
2. DEFINE an ARRAY of program records. The members of the records must match the fields of the previously specified screen array in order, number and data types
3. Open and display the form that contains the specified screen array (you may use the OPEN FORM and then the DISPLAY FORM statements or the OPEN WINDOW ... WITH FORM statement)
4. Populate the program array with the data to be displayed
5. Call the set_count(i) built-in function. Here, "i" is the number of the records filled with data
6. Use the DISPLAY ARRAY statement to display these data to a screen array

The set_count() function specifies the initial value for arr_count() function. If it is not set, no data will be displayed. For detailed information, see the Querix reference on built-in functions.

The data types of the values in each record should correspond to the data types of the fields of the screen array. The size of screen array determines the number of program records 4GL can display at a time. The



size of a program array determines the number of data it can store. If the size of a program array is larger than the size of a screen array, the user can scroll through the rows in the form.

The DISPLAY ARRAY statement has the following effect:

1. The values of the program array are displayed to the fields of the screen array
2. The cursor is moved to the first field of the first screen record
3. The user is expected to press either scroll key or accept key.

The DISPLAY ARRAY statement is not terminated until the Accept or Interrupt key is pressed, unless it includes the WITHOUT SCROLL keywords.

The values are displayed by means of the DISPLAY ARRAY statement in the same way as by the DISPLAY statement:

- Numbers are right-justified, if a number does not fit the field length, an asterisk is put in the field instead. (currently 4GL does not display anything)
- Literal strings and TEXT values are left-justified, if a literal string is longer than the field, the superfluous symbols are truncated from the right
- BYTE values cannot be displayed with the help of DISPLAY statement, the field will display <byte value> instead. To display a value of BYTE data type use PROGRAM statement

Binding Clause

The Binding clause of the DISPLAY ARRAY statement is always present and it has the following structure:



Element	Description
Program Array	It can be the name of a variable of ARRAY or DYNAMIC ARRAY data type without any additional extensions
Screen Array	The name of a screen array followed by asterisk (e.g. screen_array.*)

DISPLAY ARRAY statement can be used to display both dynamic and static arrays in the same way. The Binding clause is the only obligatory clause for this statement. E.g.:

```
DISPLAY ARRAY my_dyn_arr TO form_scr_arr.*
```

The ATTRIBUTE Clause

The ATTRIBUTE clause of the DISPLAY ARRAY statement has the structure of the common [ATTRIBUTE clause](#). All the attributes specified in it are applied to the fields within the screen array, except for the CURRENT ROW DISPLAY attribute. The following example displays all the items within the screen array *sa_account* in BLUE:

```
DISPLAY ARRAY l_account_arr TO sa_account.*  
ATTRIBUTES (BLUE)
```

You can use both singular (ATTRIBUTE) and plural (ATTRIBUTES) forms of the keyword as synonyms.



All the other attributes applied to these fields (by OPTIONS statement, OPEN WINDOW ATTRIBUTE clause, etc.) are overridden. The INVISIBLE attribute is ignored while the DISPLAY ARRAY statement is in effect. The ATTRIBUTE clause of the DISPLAY ARRAY statement can include special attributes which are not included into standard ATTRIBUTE clause.

The CURRENT ROW DISPLAY Attribute

To highlight the current row of the screen array, you may use the special attribute CURRENT ROW DISPLAY which is valid in the ATTRIBUTE clause of the DISPLAY ARRAY statement but not in the DISPLAY statement. It has the following syntax:

ATTRIBUTE (CURRENT ROW DISPLAY = “*attribute, attribute*”)

The attributes enclosed in the quotation marks apply only to the currently highlighted row of the screen array. The following example highlights the current row of the *sa_account* screen array:

```
DISPLAY ARRAY l_account_arr TO sa_account.*  
ATTRIBUTES (CURRENT ROW DISPLAY = "WHITE, REVERSE")
```

There should be at least one attribute within the quotation marks.

The HELP Clause

The HELP clause is an optional clause that specifies the number of the help message that is associated with the DISPLAY ARRAY statement.



Element	Description
Integer Expression	A 4GL expression that returns a positive integer greater than 0 and less than 32,767.

The message defined in this clause appears in the help message window when the user presses the Help key while the DISPLAY ARRAY statement is in effect, that is until the Accept key is not pressed. The Help key may be defined in the OPTIONS statement in the body of the program, by default CONTROL-W is the help key.

```
DISPLAY ARRAY my_arr TO scr_arr.*  
HELP 12
```

The integer expression identifies the message within the help file. The help file which contains the help message must be specified in the HELP FILE clause of the OPTIONS statement that precedes the DISPLAY ARRAY statement. If 4GL cannot open the specified help file, if the indicated number is greater than 32,767, or if the number is absent from the help file an error message will appear. The **qmsg** command can be used in the command line environment to create a compiled version of the help file.



	Note: A non-compiled help file is a message file with the extension .msg (help_file.msg). Only a compiled message file can be referenced as a help file. Notice that a compiled help file has another extention, e.g. .erm (help_file.erm).
---	--

The WITHOUT SCROLL Clause

This optional clause consists of the WITHOUT SCROLL keywords. When 4GL encounters the WITHOUT SCROLL keywords, it takes the following steps:

- Displays the array.
- Transfers the program control to the statement following the DISPLAY ARRAY statement without waiting for the user to press the Accept key.
- Disables scroll keys.
- Skips all the optional clauses in the DISPLAY ARRAY statement except the ATTRIBUTE clause:
 - The keys and actions in the ON KEY and ON ACTION clauses of the DISPLAY CONTROL clause have no effect when pressed.
 - The BEFORE DISPLAY and AFTER DISPLAY clauses are not executed.
 - The HELP clause is ignored.

Optional clauses have no effect even if they are specified, because 4GL does not wait for the user to press the Accept key, it leaves the DISPLAY ARRAY statement immediately after the WITHOUT SCROLL keywords have been encountered.

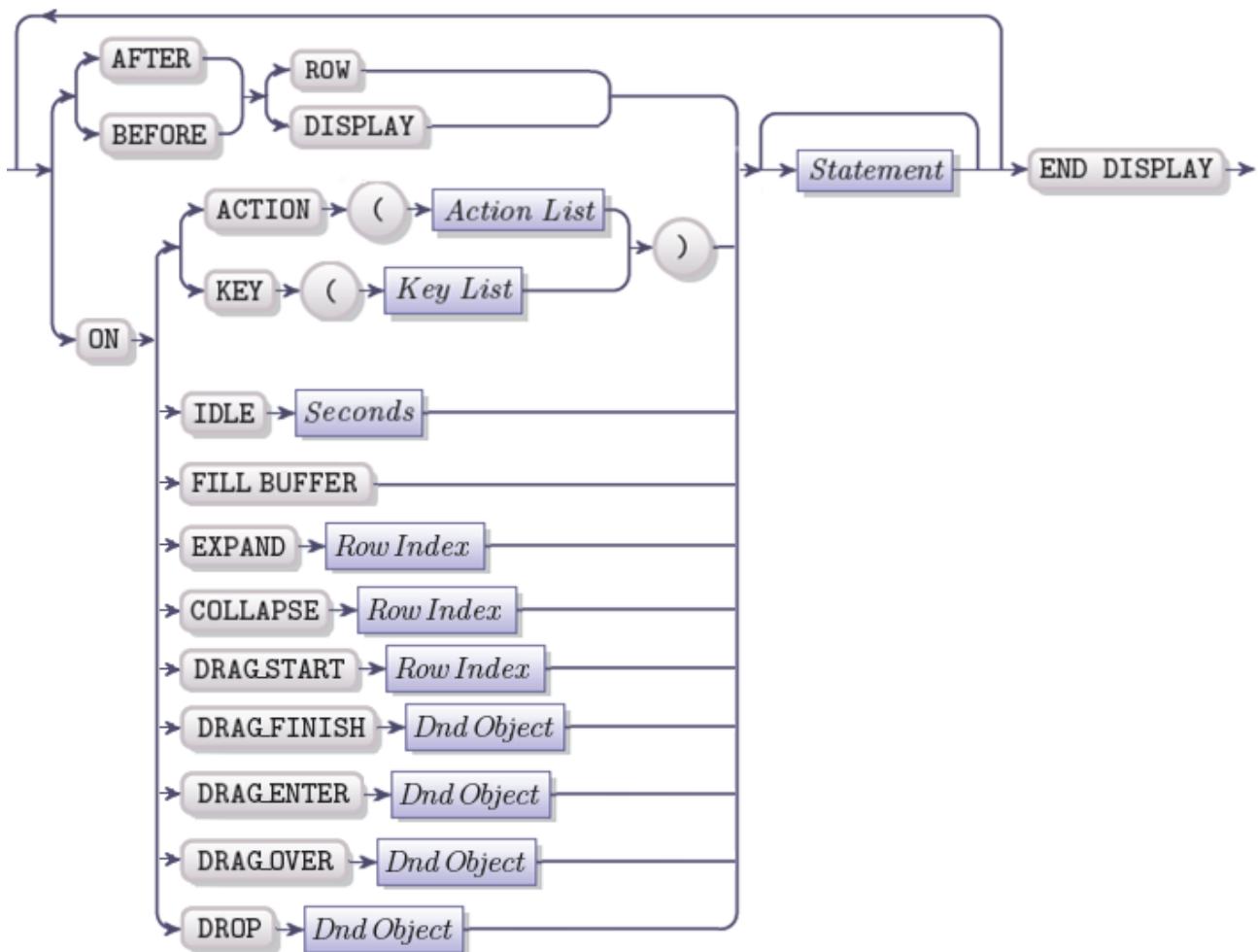
In the example below, the green colour will be applied to the displayed array, but the user will not be able to view the help message:

```
DISPLAY ARRAY my_arr TO scr_arr.*  
ATTRIBUTE (GREEN)  
HELP 12  
WITHOUT SCROLL  
PROMPT "Enter the reason " FOR reas
```

The array will still be displayed on the screen after the control is passed to the PROMPT statement, unless this statement displays something else to the form fields or moves to another 4GL window. You cannot use scroll keys to scroll the array displayed without scroll. However, in the GUI clients you can scroll the array by means of the mouse cursor

The DISPLAY Control Clause

The optional DISPLAY control clause has the following structure:



Element	Description
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks
Key List	A list that consists of one to four key names separated by commas
Statement	The Statement block of the DISPLAY control clause
Seconds	An integer value specifying a delay time in seconds

The END DISPLAY keywords are required only if the DISPLAY ARRAY statement has at least one DISPLAY control clause.

The DISPLAY ARRAY statement called within the DIALOG instruction supports Drag and Drop events that give the user the following abilities:

- Drag the objects from lists and tree-views to other lists and tree-views within the same 4GL forms and programs.
- Drag the objects from lists and tree-views to other lists and tree-views within different 4GL forms and programs.
- Drag the objects from lists and tree-views to other lists and tree-views between 4GL applications and other desktop programs.

The available Drag and Drop behaviour is described in corresponding sections.



Statement Block

The Statement block of the DISPLAY ARRAY clause can include:

- Any 4GL executable statement
- SQL statements that can be embedded into 4GL code
- ACCEPT DISPLAY keywords
- NEXT FIELD keywords
- CONTINUE DISPLAY keywords
- EXIT DISPLAY keywords
- BEGIN ... END logical block (for the details see the "[Logical Blocks](#)" section of this reference)

The BEFORE DISPLAY Clause

The BEFORE DISPLAY clause specifies the statements that are to be executed before any information is displayed to the screen array. A DISPLAY ARRAY statement can contain only one BEFORE DISPLAY clause. The following example of the BEFORE DISPLAY clause is used to display a specified row of the program array at a specified line of the screen array with the help of a built-in function:

```
BEFORE DISPLAY
CALL fgl_dialog_setcurrline(5,current_row)
```

Here the *current_row* is a variable of integer data type.

The BEFORE ROW Clause

The BEFORE ROW clause can be included into the DISPLAY ARRAY statement, if you want 4GL to perform certain actions before the cursor moves to another row. Here, a row means a line within a screen array. Only one BEFORE ROW block can be included into a DISPLAY ARRAY statement.

In the following example each time the cursor is moved to another line within the screen array, the value of the current line is assigned to variable *i* and the value of the array element *l_account_arr.acc_id* on the line [*i*] is assigned to variable *var1*

```
DISPLAY ARRAY l_account_arr TO sa_account.*
BEFORE ROW
LET i = arr_curr()
LET var1 = l_account_arr[i].acc_id
```

The ON KEY Clauses

The keys can be assigned to form widgets and the actions will be triggered when such widget is pressed. However, if a key combination is not assigned to any widget, it can be activated with the help of the keyboard.

One DISPLAY ARRAY statement can contain any number of the ON KEY clauses, they can be placed in any order. An ON KEY clause can use one or several of the following keys separated by commas and enclosed in parentheses to specify the key to which the statements in the ON KEY clause refer:

ESC/ESCAPE	ACCEPT
NEXT/ NEXTPAGE	PREVIOUS/PREVPAGE
INSERT	DELETE



RETURN	TAB
INTERRUPT	HELP
LEFT	RIGHT
UP	DOWN
F1 – F256	CONTROL- <i>char</i>

Any character can be used as *char* in the combination CONTROL-*char* except the following characters: A, D, H, I, J, K, L, M, R, and X. The key names can be entered either in lower case or in upper case letters.

The Statement block following the ON KEY clause is executed when the key specified in this ON KEY clause is pressed. The form where the DISPLAY ARRAY statement takes place is deactivated, though the statement is not terminated and the form will be reactivated, when 4GL finishes executing the ON KEY clause, unless it contains the EXIT DISPLAY statement. If an ON KEY clause contains the EXIT DISPLAY statement, the DISPLAY ARRAY statement will be terminated.

Some keys require special attention if used in the ON KEY block:

Key	Usage Features
ESC/ESCAPE	If you want to use this key in an ON KEY block, you must specify another key as the Accept key in the OPTIONS statement, because ESCAPE is the Accept key by default
INTERRUPT	DEFER INTERRUPT statement must be executed if you want to use this key in an ON KEY block. On pressing the INTERRUPT key the corresponding ON KEY block is executed, int_flag is set to non-zero but the DISPLAY ARRAY statement is not terminated
QUIT	DEFER QUIT statement must be executed in order that this key could be used in the ON KEY block. On pressing the QUIT key the corresponding ON KEY block is executed and int_flag is set to non-zero
CTRL- <i>char</i> (A, D, H, L, R, X)	4GL reserves these keys for field editing and they should not be used in the ON KEY block
CTRL- <i>char</i> (I, J, M)	These key combinations by default mean TAB, NEWLINE and RETURN. If they are used in the ON KEY block, they cannot perform their default functions while the OK KEY block is functional. Thus, if you use these keys in the ON KEY block, the period of time during which this block is functional should be restricted.

Some restrictions in key usage might be applied depending on your operational system. Many systems use such key combinations as CONTROL-C, CONTROL-Q, and CONTROL-S for the Interrupt, XON, and XOFF signals.

After all the statements within the ON KEY clause have been executed, the form with the screen array is reactivated, the cursor is in the same location where it has been before the key specified in an ON KEY block has been pressed, unless the ON KEY clause contains the EXIT DISPLAY keywords, which make 4GL execute the first statement that follows the END DISPLAY statement.

The following example specifies two keys that have the same effect when pressed during the DISPLAY ARRAY statement:

```
ON KEY (ACCEPT, F11)
CALL my_func()
```



The ON ACTION Clauses

The actions which you have assigned to the buttons in the form specification file can be used in an ON ACTION clause. The ON ACTION clauses are treated by 4GL very much like the ON KEY clauses. 4GL executes the statements specified in it, when the user presses a button or other widget on the form to which the corresponding action is assigned. Whereas an ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard, an ON ACTION clause is activated with the help of a form widget.

An event (action) can be assigned to the following widgets: button, radio button, check box, function field, and hotlink. It must be entered into the corresponding field in the Graphical Form Editor without quotation marks and white spaces. In the text form editor, it must be enclosed into quotation marks:

```
CONFIG = "action {Name of the Button}"
```

The actions in an ON ACTION clause must be represented by a character string enclosed both in quotation marks and in parentheses and it must be the same as the name of the action in the form specification file:

```
ON ACTION ("action", "alternative_event")
```

Lycia II also supports an in-built action named "*doubleclick*". The statements specified in the ON ACTION ("doubleclick") control block are invoked when the user performs a double click of the left mouse button at any place of the application window.

```
ON ACTION ("doubleclick")
```

It is also possible to set different doubleclick actions for different form widgets. To do this, one should add the INFIELD clause and specify the field or fields, double clicking on which should invoke the statements in the ON ACTION block:

```
ON ACTION ("doubleclick") INFIELD (f001)
```

There can be any number of the ON ACTION clauses in a DISPLAY ARRAY statement. Up to four actions enclosed in quotation marks and separated by commas may be included into one ON ACTION clause. As well as with the ON KEY clause, any 4GL or SQL statements can be included into an ON ACTION clause. The statements of a corresponding ON ACTION clause are executed when the widget within a screen form to which the action is assigned is pressed. The form where the DISPLAY ARRAY statement takes place is deactivated while the statements are executed, though the statement is not terminated and the form will be reactivated, when 4GL finishes executing the ON ACTION clause, unless it contains the EXIT DISPLAY statement. If the ON ACTION clause contains the EXIT DISPLAY statement, the DISPLAY ARRAY statement will be terminated.

The example below represents an ON ACTION clause which is triggered when the button with the "update" event is pressed.

```
ON ACTION ("update")
CALL update(a,b)
```



The ON IDLE Clauses



Element	Description
Seconds	An integer value specifying the time period in seconds

The ON IDLE clause is used to specify the actions that will be taken if the user performs no actions during the specified period of time.

The *seconds* parameter should be represented by an integer value or a variable which contains such a value. If the value is specified as zero, the input timeout is disabled. It is advised that you specify the *seconds* parameter as a relatively long period of time (more than 30 seconds), because shorter delays may be caused by some external situations that distract the user from the application, so, the control block will be executed inappropriately:

```

DISPLAY ARRAY my_arr TO scr_arr.*
ON IDLE 120
CALL fgl_message_box("Yoy've been out for 2 minutes")
END INPUT

```

The ON FILL BUFFER Clause

The ON FILL BUFFER clause is activated, when the cursor goes beyond the last line of the currently displayed dynamic array.

The clause is used to add rows to a currently displayed program array, basing on the number of rows and array offset .To get the number of rows, use fgl_dialog_getbufferlength() function. To get the offset, use the fgl_dialog_getbufferstart() built-in function.

Typically, before an array is displayed, the total number of rows is specified by the set_count() function or the COUNT option. The program will execute the ON FILL BUFFER clause till the number of rows which is less than expected for the page, is specified, or till the total rows number is set to other then -1 with the ui.Dialog.setArrayLength() method.

```

DISPLAY ARRAY prog_arr TO srec.* ATTRIBUTES(COUNT=-1)
ON FILL BUFFER
    LET arr_offs = fgl_dialog_getBufferStart()
    LET arr_length = fgl_dialog_getBufferLength()
    LET row_ix = arr_offs
    FOR i=1 TO arr_length
        FETCH ABSOLUTE row_ix c1 INTO prog_arr[i].*
        IF SQLCA.SQLCODE!=0 THEN
            CALL DIALOG.setArrayLength("srec",row-1)
            EXIT FOR
        END IF
        LET row_ix = row_ix + 1
    END FOR

```



The ON EXPAND Clause

When working with trees that contain big number of nodes and only first tree levels are displayed, there is no need to pass all the possible nodes to the program array. ON EXPAND and ON COLLAPSE clauses are used to modify tree structure on demand.

The ON EXPAND clause is triggered after the treeview node is opened (expanded). The block is usually specified when it is necessary to create dynamic trees, in which nodes addition depends on the nodes that the final user opens. The block needs the row specification:

```
DISPLAY ARRAY mytree TO tree_scr.* ATTRIBUTES (UNBUFFERED)
    ON EXPAND (row_ix)
        DISPLAY "You expended row #", row_ix
        -- Fill mytree[row_ix] with children nodes
```

The ON COLLAPSE Clause

The ON COLLAPSE clause is triggered when the treeview node is closed (collapsed). The block is usually specified when it is necessary to create dynamic trees, in which nodes addition depends on the nodes that the final user opens. The block needs the row specification:

```
DISPLAY ARRAY mytree TO tree_scr.* ATTRIBUTES (UNBUFFERED)
    ON COLLAPSE (row_i)
        DISPLAY "You collapsed row #", row_ix
        -- Remove children nodes from mytree [row_ix]
    END DISPLAY
```

DRAG and DROP control clauses

The Drag and Drop events are activated by a special set of control blocks. These blocks are activated when the user performs corresponding actions.

All the Drag and Drop clauses need a ui.DragDrop object as a parameter. This object is to be declared as a variable before the Drag and Drop actions start.

By default, the dragged object is copied to the drag and drop buffer as a tab-separated list of values. You can change it using the SetMimeType() and SetBuffer() functions. These functions, as well as the others mentioned in this block, are described in the Lycia Built-in Functions guide.

The order of the Drag and Drop control blocks execution, the corresponding triggering actions and the dialog in which these actions are performed are the following:

Control block	Dialog	User actions
ON DRAG_START	The user starts dragging an object from the source dialog	Source dialog
ON DRAG_ENTER	The user sets the mouse cursor to the drop target dialog	Target dialog
ON DRAG_OVER	The mouse moves from row to row within the target dialog or user changes the drop operation (from move to copy or vice versa)	Target dialog
ON DROP	When the user drops the dragged object by releasing the mouse button over the target dialog	Target dialog
ON DRAG_FINISHED		Source dialog



The DRAG_START Clause

The ON DRAG_START clause is performed when the user starts the drag operation. The dragging is cancelled for the current dialog, if there is no dialog trigger.

The dragged object can be copied or moved to the destination. In the first case, the copied object remains in the drag source dialog; in the second case, the dragged object is deleted from the source dialog after it is inserted to the destination place. The ON DRAG_START block is used to specify the operation to be executed by the Drag and Drop action.

The default and only possible Drag and Drop operation is specified by Set.Operation() method. The AddPossibleOperation() method can be applied to allow the user to choose himself whether the data should be copied or pasted to.

If you need to allow the user drop the dragged object outside the 4GL application, you have to specify the MIME type with the SetMimeType() and SetBuffer() methods.

Below is given an example of the ON DRAG_START control block specification:

```
DEFINE dd ui.DragDrop
...
DISPLAY ARRAY pr_arr TO scr_arr.* ...
...
ON DRAG_START (dd)
    CALL dd.setOperation("copy") -- Copy is the default operation
    CALL dd.addPossibleOperation("move") -- User can toggle to move if
needed
    CALL dd.setMimeType("text/plain")
    CALL dd.setBuffer(pr_arr[arr_curr()].addrs)
...
END DISPLAY
```

The DRAG_FINISHED Clause

The ON DRAG_FINISHED clause is an optional block that is executed when the drop operation is performed or cancelled.

In this control block, one should use the GetOperation() method in order to retrieve the type of operation that is performed when the user drops the dragged object. If the operation is completed successfully, the method returns "move" or "copy". If the method returns NULL, it means that the drop operation was denied, and ON DRAG_FINISHED block can be ignored.

If the dragged object is to be moved from the current DISPLAY ARRAY to another one, the ON DRAG_FINISHED block should contain the commands that will delete the dragged object from the drag source after it is copied to the destination place.

```
DEFINE dd ui.DragDrop
...
DISPLAY ARRAY pr_arr TO scr_arr.* ...
...
ON DRAG_START (dd)
    LET last_drow = arr_curr()
...

```



```
ON DRAG_FINISHED (dd)
    IF dd.getOperation() MATCHES "move" THEN
        CALL DIALOG.deleteRow(last_drow)
    END IF
    ...
END DISPLAY
```

The ON DRAG_ENTER Clause

The ON DRAG_ENTER block is triggered when the mouse cursor with the dragged object enters the area of the drop target dialog, if the ON DROP clause is specified.

If the ON DRAG_ENTER control block is not specified, the entering to the target dialog is accepted by default. However, you should remember, that if you specify the ON DROP block, you should also give the ON DRAG_ENTER to make the program cancel the drop if the dropped objects are dragged from other applications and are of an unsupported MIME type.

The SelectMimeType() method is used to check whether the MIME type of the dragged object is available in the Drag and Drop buffer. You should pass a MIME type as the argument to this method, and it will return TRUE if the given type is used. If you need to check several MIME types, you can call the SelectMimeType() method for several times.

You can allow the drop under some specific circumstances using the SetOperation() method, which will deny the operation if you pass NULL as its argument.

You can also use the SetFeedback() method within the ON DRAG_ENTER block in order to specify visual effect that will be caused by the cursor placed over the drop target object.

There are three possible values that can be passed as an argument to the SetFeedback() method. They are:

1. "insert"
2. "select"
3. "all"

```
DEFINE dd ui.DragDrop
    ...
DISPLAY ARRAY pr_arr TO scr_arr.* ...
    ...
ON DRAG_ENTER (dd)
    IF dd.selectMimeType("text/plain") THEN
        CALL dd.setOperation("move")
        CALL dd.setFeedback("all")
    ELSE
        CALL dd.setOperation(NULL)
    END IF
ON DROP (dd)
    ...
END DISPLAY
```



The ON DRAG_OVER Clause

If the ON DROP block is defined, the ON DRAG_OVER block is activated after the ON DRAG_ENTER one, during the cursor movement over the drop target or when the Drag and Drop operation is switched from copy to move or vice versa.

The block is optional and can be necessary in cases when the acceptance of the drop or the drop operation depends on the drop target row.

The ON DRAG_OVER block is performed only once for each row and is not executed if the drop was denied within the ON DRAG_ENTER block by SetOperation(NULL) method.

In the ON DRAG_OVER block you can return the index of the current row of the target array by means of the GetLocationRow() method. The method returns an integer number and doesn't need any arguments.

In the TreeView, you can use the GetLocationParent() method to return the integer index of the parent node of the child node to be inserted and to see if the dropped object will be inserted as a child or as a parent.

You can deny or allow the drop using the results of this check.

The ON DRAG_OVER block can be executed even if the mouse does not change the position, because the current drag and drop operation is selected in previously performed ON DRAG_ENTER and ON DRAG_OVER events. You can check the current operation with the GetOperation() and change it with the SetOperation() method.

If the SetOperation(NULL) method was used in a previous ON DRAG_OVER event to deny a drop, you can reset the drag and drop operation using the SetOperation("move"|"copy") method.

Below is given an example of the ON DRAG_OVER block specification:

```
DEFINE dd ui.DragDrop
...
DISPLAY ARRAY pr_arr TO scr_arr.* ...
...
ON DRAG_ENTER (dd)
...
ON DRAG_OVER (dd)
    IF pr_arr[dd.getLocationRow()].copy_accept THEN
        CALL dd.setOperation("copy")
    ELSE
        CALL dd.setOperation(NULL)
    END IF
ON DROP (dd)
...
END DISPLAY
```

In cases, when the user is allowed to choose between copying and moving the data, you can use the ON DRAG_OVER block to set up different program behavior for each case:

```
ON DRAG_OVER (dd)
CASE dd.getOperation()
WHEN "move"
MESSAGE "You're moving the object to row ", dd.getLocationRow()
WHEN "copy"
MESSAGE "You're copying the object to row ", dd.getLocationRow()
```



```
END CASE
```

The ON DROP Clause

The target list will not accept the dropped object if you don't specify the ON DROP control block, which is executed after the user releases the mouse button over the target list thus dropping the dragged object. The block is not executed, if the drop was previously denied by SetOperation(NULL) method called in ON DRAG_OVER and ON DRAG_ENTER clauses.

The ON DROP block is also used to access the Drag and Drop buffer and to check the MIME type if the data are to be copied from external applications. One needs to call the GetSelectedMimeType() to check the MIME type and then call the GetBuffer() method to return the data stored in the buffer.

The GetLocationRow() method used in this block, returns the index of the target array row, and you can use it to pass the drop object to the row that the user selected:

```
ON DROP (dd)
LET pr_arr[dd.getLocationRow()].indices == dd.getBuffer()
...
```

If you want to allow the drag and drop within one and the same dialog, you can add the DropInternal() method to the ON DROP block. The method does not need any arguments and does not return any value.

```
DISPLAY ARRAY arr_tree TO sr_tree.* ...
...
ON DROP (dnd)
CALL dnd.dropInternal()
```

To specify a drop operation for a tree-view, you should take the following steps:

1. Call the GetLocationParent() method to find out whether the object was dropped as a sibling or as a child node. The method will return the node index.
2. Call the GetLocationRow() method to return an integer number identifying the target row of the target dialog.
3. If the row detected by step 2 is the child of the parent detected by step 1, the new row should be inserted before the GetLocationRow(). Otherwise, add the new row as a child of the parent node found by the GetLocationParent() method.

The AFTER ROW Clause

The AFTER ROW clause can be included into the DISPLAY ARRAY statement, if you want 4GL to perform certain actions each time the cursor leaves a row. Here, a row means a line within a screen array. Only one AFTER ROW block can be included into a DISPLAY ARRAY statement.

In the following example, each time the cursor leaves a row of the screen array and moves to another row, the notification is displayed to the Output console:

```
DISPLAY ARRAY my_arr TO sa_arr.*
AFTER ROW
DISPLAY "You leave the current row"
```



The AFTER DISPLAY Clause

The AFTER DISPLAY clause is executed before the DISPLAY ARRAY statement is terminated. If the AFTER DISPLAY clause is present, the statements in this clause will be executed before the DISPLAY ARRAY statement is terminated and before 4GL starts executing the first statement after the END DISPLAY clause. There can be only one such clause within the DISPLAY ARRAY statement.

The AFTER DISPLAY clause is executed when the user presses:

- Accept key
- Interrupt key (if the DEFER INTERRUPT statement is in the effect)
- Quit key (if the DEFER QUIT statement is in the effect)

The AFTER DISPLAY clause is not executed in the following cases:

- The Interrupt key is pressed with no DEFER INTERRUPT statement in effect
- The Quit key is pressed with no DEFER QUIT statement in effect
- If the EXIT DISPLAY statement has been executed in any optional clause

Here is a sample of the AFTER DISPLAY clause:

```
DISPLAY ARRAY
...
AFTER DISPLAY
MESSAGE "The display will now be terminated."
SLEEP 3
END DISPLAY
```

The built-in functionality

The DISPLAY ARRAY supports some built-in functionality which allows to work with the data displayed to the screen array or to the tree view.

This functionality is bound to the screen grid and does not need any additional coding.

The Built-In Sorting

If the dialog is performed in a tree-view or a table, the user can sort the rows.

To sort the rows of a list, the user should click on the header of the column by which the sorting should be performed. A click on the header invokes a GUI event that makes the runtime system change the order of the rows displayed to the table.

Note, that the rows are sorted only visually and the data in the program array remain untouched. Therefore, after the rows were sorted this way, the current index of a row and its visual position may be different.

The system uses the standard collation to order the rows, and its behaviour depends on the locale settings. Therefore, the resulting order can vary from the order generated on DATABASE server with the ORDER BY clause of the SELECT statement.

The table data cannot be sorted if you use the DISPLAY ARRAY in the paged mode, because in this case not all the resulting rows are known to the system.

When the user or the program closes the application window, the information about the sort column and order is passed to the user settings database of the system. This information will be retrieved and applied to the table when the window is opened again. Therefore, when the program restarts, the rows will be sorted according to the last settings. The saved settings are specific for each list container.



By default, the built-in sorting is enabled, but you can switch it off for a table or a tree by adding the UNSORTABLECOLUMNS attribute to the list container, or the UNSORTABLE attribute for the column/field. The UNSORTABLECOLUMNS attribute may be useful in the lists engaged in the INPUT ARRAY dialog.

The Built-In Find

The DISPLAY ARRAY statement supports the default built-in find feature, which includes implicitly invoked *find* and *findnext* actions. These actions can be treated as the regular ones.

The find action is triggered with the CONTROL-F keypress by default. The search can be performed in any list container (tables, trees, or grids). When the action is triggered, the program opens a popup dialog window to which the user should enter the search value. When the user enters the value and presses the OK button, the dialog starts searching a row in which a field value would match the entered one. After the first value is found, the user can press CONTROL-G (by default) to call the *findnext* action and to find the next matching in the list. The *findnext* action does not open the search dialog window, but resumes the search using the previously entered value.

The search value is compared with the string displayed to the fields, i.e. is based on the value format. For example, MONEY fields will have the currency symbol and the user should enter the search value including this symbol and using correct separators.

By default, the search is performed in all the columns, but the find dialog box offers the user to choose a specific column for search.

The search scans only the text widgets. Columns, using radio buttons, Checkboxes, Images, as well as TEXT and BYTE fields are not processed during the search. It is also important to remember, that the program searches among the values stored in the memory, so the search is disabled when the application performs a display in the paged mode (ON FILL BUFFER is used). In dynamic tree views, the built-in find searches among the nodes stored in the program array.

You can specify your own actions for the *find* and *findnext* actions using the ON ACTION clause. In this case, the default built-in find action is disabled.

The Built-In Seek

When the user inputs alphabetic characters during the DISPLAY ARRAY, the system automatically seeks for the next row with a field having the value which starts with the inputted characters. The seek restarts from the current row when the user inputs new characters.

The seek action is applied to any type of list containers (table, grid, treeview); it is not performed for date/time, numeric and large data columns. Only character fields are involved into the quick search. By default, the quick search scans all the fields, but if the list was sorted, the system considers the sort column to be the most important and involves only this column into search.

The seek search is not case sensitive. The user can quickly type several characters and then search for a value starting with the inputted letters. The new search filter is applied very quickly, in a timeout which is less than a second.

If the typed characters do not match any of the values present in the list, the *not found* error is displayed automatically.

The quick search is disabled, if some program action is activated by an alphabetic character.

Note, that the search is scanning only the rows that are currently stored in the memory. Therefore, the seek is disabled when the DISPLAY ARRAY is performed in the page mode. For tree-views, the seek will be applied only to the nodes present in the program array.

The ACCEPT DISPLAY Keywords

The ACCEPT DISPLAY keywords are used for the DISPLAY ARRAY statement validation. On doing this, ACCEPT DISPLAY passes a program control to the AFTER DISPLAY control block exiting the DISPLAY ARRAY statement.

The execution of the statements following ACCEPT DISPLAY will not take place:



```
# Only the first DISPLAY will be executed

ON KEY (F10)
    DISPLAY "+"
    ACCEPT DISPLAY
    DISPLAY "-"
```

The CONTINUE DISPLAY Keywords

The CONTINUE DISPLAY keywords pass a program control from a control block back to the DISPLAY ARRAY statement. The cursor is returned to the first field of the first row of the screen array. It is useful when you want to return from a deeply nested loop and continue displaying the array.

After these keywords all subsequent statements in the current control block of the DISPLAY ARRAY statement are skipped. The array is redisplayed. However, if one of the previous control blocks has displayed other values to one or more fields of the screen array, these values will continue being displayed and the initial values will not be restored.

The EXIT DISPLAY Keywords

The EXIT DISPLAY keywords are used to terminate the DISPLAY ARRAY statement. They can be included into any of the DISPLAY control clauses. When 4GL encounters these keywords, it skips all the statements that are located between the EXIT DISPLAY and the END DISPLAY statements and passes control to the first statement that follows the END DISPLAY statement.

```
ON ACTION ("EXIT")
    EXIT DISPLAY
```

The END DISPLAY Keywords

The END DISPLAY keywords terminate the DISPLAY ARRAY statement. These keywords are required when:

- A detached DISPLAY ARRAY statement has at least one DISPLAY control clause
- A DISPLAY ARRAY statement without any optional clauses is used in a DISPLAY control clause of another DISPLAY ARRAY statement or in the input control clause of a PROMPT statement
- A DISPLAY ARRAY statement without any optional clauses is a part of a CONSTRUCT, INPUT, or INPUT ARRAY statement, if the ON KEY (or ON ACTION) clause of the enclosing statement follows the enclosed DISPLAY ARRAY statement. If the END DISPLAY keywords were absent from the example below, it would have been not clear to which statement the ON KEY clause refer: to the DISPLAY ARRAY statement or to the INPUT/CONSTRUCT statement:

```
INPUT BY NAME order_id.*
    BEFORE INPUT
        DISPLAY ARRAY ord_arr TO ord_scr.*
    END DISPLAY
    ON KEY (F12)
    ...
END INPUT
```



The END DISPLAY keywords are not required if a DISPLAY ARRAY statement is used independently without any DISPLAY control clauses.

Scrolling when the DISPLAY ARRAY Statement is in Effect

To scroll through the screen array that is currently used by the DISPLAY ARRAY statement you can use the following keys:

- Down or Right Arrow – moves the cursor down one row. If the cursor has been on the last row of the screen array, but there are still rows left in the program array below it, 4GL moves the information up one row. If the cursor is at the last row of the program array and the screen array, the message will appear which informs the user that there are no more rows in this direction.
- Up or Left Arrow - moves the cursor up one row. If the cursor has been on the first row of the screen array, but there are still rows left in the program array above it, 4GL moves the information down one row. If the cursor is at the first row of the program array and the screen array, the message will appear which informs the user that there are no more rows in this direction.
- F3 – scrolls to the next full page of the program records, this is the default NEXT key. This can be changed in the OPTIONS statement.
- F4 - scrolls to the previous full page of the program records, this is the default PREVIOUS key. This can be changed in the OPTIONS statement.

If you declare the DISPLAY ARRAY ... WITHOUT SCROLL, the scroll keys are disabled. In such cases, the user can scroll through the array only in the GUI mode with the help of the mouse cursor.

Completing the DISPLAY ARRAY Statement

The DISPLAY ARRAY statement is completed when any of the following happens:

- Accept key is pressed
- Interrupt key is pressed
- Quit key is pressed
- 4GL executes the EXIT DISPLAY statement
- 4GL encounters the WITHOUT SCROLL keywords

Each of these actions deactivates the screen form in the character mode, the Interrupt and Quit keys terminate not only the DISPLAY ARRAY statement but also the program, if no DEFER INTERRUPT or DEFER QUIT statement is in effect. In the GUI mode the screen form is not deactivated and you are able to scroll though the array using the mouse cursor, unless the statements that follow the DISPLAY ARRAY statement deactivate the form, remove the displayed values or move the program control to another window.

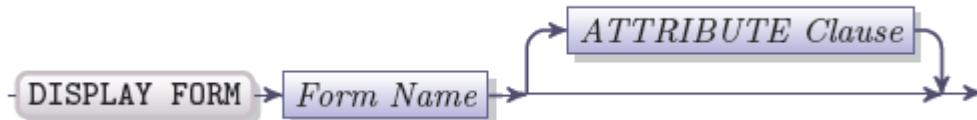
If the DEFER INTERRUPT statement is in effect, pressing the Interrupt key will result in setting the int_flag to zero and termination of the DISPLAY ARRAY statement without terminating the program.

If the DEFER QUIT statement is in effect, pressing the Quit key will result in setting the quit_flag to zero and termination of the DISPLAY ARRAY statement without terminating the program.



DISPLAY FORM

The DISPLAY FORM statement is used to display previously compiled and opened form.



Element	Description
Form name	The name of the form used previously in the OPEN FORM statement
ATTRIBUTE Clause	The optional ATTRIBUTE clause where the display attributes can be specified

To be able to display a form with the DISPLAY FORM statement, you must first open it with the help of the OPEN FORM statement, where you declare the name of the form. Use this name of the form in the DISPLAY FORM statement. If you open a form with the help of the OPEN WINDOW with the WITH FORM clause, you need neither OPEN FORM nor DISPLAY FORM statement.

A form displayed by means of the DISPLAY FORM statement will be displayed in the current 4GL window. If no 4GL window is open, it will be displayed in the 4GL screen.

	Note: If the size of the form displayed by the DISPLAY FORM statement is larger than the window in which it is displayed, a runtime error will occur, informing you that the window is too small to display the form
--	---

The Form Name

A form identifier previously declared in an OPEN FORM statement can be used after the DISPLAY FORM keywords to specify the form to display.

The ATTRIBUTE Clause

The ATTRIBUTE clause of the DISPLAY FORM statement has the structure of the common [ATTRIBUTE clause](#). If the ATTRIBUTE clause of the DISPLAY FORM statement contains the INVISIBLE attribute, this attribute is ignored. The attributes specified in the ATTRIBUTE clause of the DISPLAY FORM statement are applied to any fields that do not have attributes specified in the ATTRIBUTES section of a form specification file, or in the subsequent OPTIONS statement. If a form is displayed in a 4GL window, the attributes of the DISPLAY FORM statement override the colour attributes of the window. In their turn the attributes of the DISPLAY FORM statement can be overridden by the attributes of a CONSTRUCT, DISPLAY or DISPLAY ARRAY statement applied to this form.



The Reserved Lines

The form is displayed at the line that has been specified as the FORM LINE within the most recent OPTIONS statement or in the corresponding OPEN WINDOW statement. The default FORM LINE is 3.

There are several lines in each 4GL window and in the 4GL screen which are reserved for specific 4GL actions. By default, they are:

Default Line Position	Reserved for displaying:
First line	<ul style="list-style-type: none">the PROMPT statement outputthe first menu line containing the menu option names
Second line	<ul style="list-style-type: none">the MESSAGE statement outputthe second menu line containing the descriptions of the menu option names
Third line	<ul style="list-style-type: none">the FORM LINE – the position of the first line of a form
Last but one line	<ul style="list-style-type: none">the COMMENT line in the graphical display mode
Last line	<ul style="list-style-type: none">the ERROR statement outputthe COMMENT line in the character display mode

The following example displays form with the name *orders* declared previously by the OPEN FORM statement:

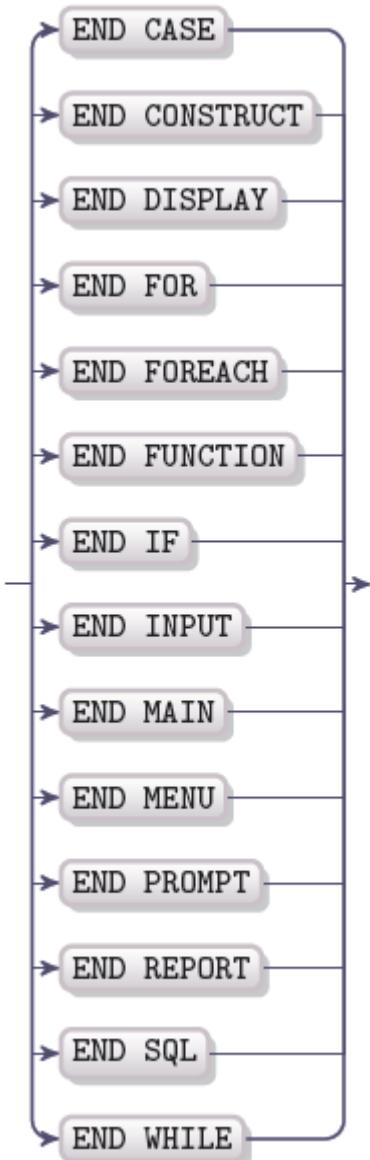
```
OPEN FORM orders FROM "ord_info"  
DISPLAY FORM orders
```

The default location of the reserved lines can be changed by means of the OPTIONS statement. You can also change the position of the form line in the ATTRIBUTE clause of the OPEN WINDOW statement that opens a window where the form is to be displayed.



END

The END keyword serves as the first part of the statements that indicate the end of compound 4GL statements. It has the structure END *keyword* where the *keyword* is the name of the 4GL statement that is to be terminated. The diagram below illustrates the statements with which the END statement is used:



The END statement occurs at the last line of a compound 4GL statement – that is a 4GL statement that has additional clauses in it. It indicates that the statement is fully executed and completed rather than terminated (for termination of a compound statement when not all its elements are yet executed, use the EXIT keyword).



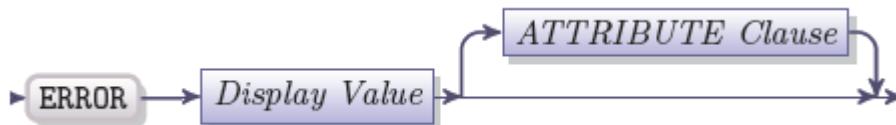
To delimit the DISPLAY ARRAY statement the END DISPLAY keywords are used. To delimit both INPUT and INPUT ARRAY statements the END INPUT keywords are used. The END RECORD keywords are used to complete the declaration of a program RECORD. For more details, see the sections on the corresponding statements in this reference. Below is an example of the END MAIN keywords demarcating the MAIN program block:

```
MAIN
...
END MAIN
```



ERROR

The ERROR statement is used to display a message at the line reserved for error messages, it is usually the last line of a 4GL window or the 4GL screen. When the ERROR statement is executed, 4GL rings the terminal bell.



Element	Description
Display Value	A variable, a quoted character string or their combination with comma separators
ATTRIBUTE Clause	Optional clause that applies display attributes to the error message displayed

The Display Value

The Display Value can consist of:

- a character string enclosed in quotation marks
- a variable of CHAR, STRING or VARCHAR data type
- a Character expression (See "[Character Expressions](#)" section for more details)

You can specify character variables and literal character strings in the error message in any combinations. The variables that follow the ERROR message, if any, will be replaced by their values, all the elements used in the error message will be concatenated. There will be no white spaces between them unless the spaces are explicitly specified.

The total length of an error message is limited to the length of a single line within a window or the 4GL screen where this message is to be displayed. The error message is cleared from the screen when any key is pressed.

You can use the CLIPPED operator to strip off the trailing blanks from variables, the USING operator to specify the format of the display, and SPACE operator to insert blank spaces, e.g.:

```
ERROR sample_patt CLIPPED, " is not valid."
```

The Error Line

The error message appears on the special line called the error line, by default it is the last line of the 4GL screen or window. If the program is run in the character mode, the error line is the last line of the 4GL screen and it is not displayed at the current window. The default position of the error line can be changed by the closest OPTIONS statement with the ERROR LINE clause. The position of the ERROR line cannot be changed by the ATTRIBUTE clause of the OPEN WINDOW statement.



The ATTRIBUTE Clause

The ATTRIBUTE clause of the ERROR statement has the structure of the general [ATTRIBUTE clause](#). The ATTRIBUTE clause of the ERROR message has some features that are not described in the general description of the ATTRIBUTE clause, they are:

- By default the REVERSE attribute is applied to the error message in character mode
- The ERROR statement ignores INVISIBLE attribute, if it is combined with any other attribute
- The ERROR statement displays the INVISIBLE attribute as NORMAL if it is the only attribute of an error message (currently it is displayed as invisible – no text)

```
ERROR "The input data is out of the allowed range"  
ATTRIBUTE (RED, UNDERLINED)
```

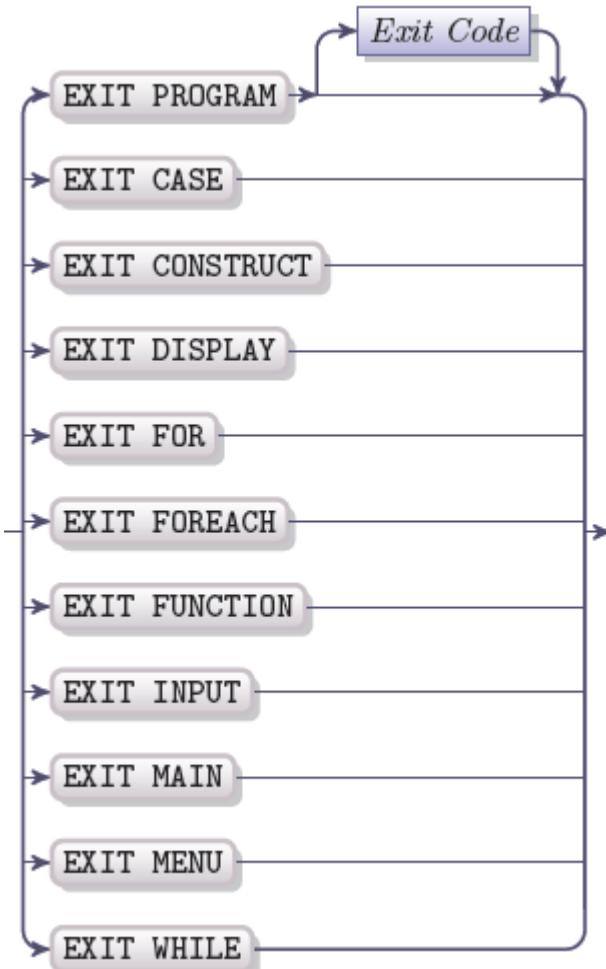
System Error Messages

The system error messages are also displayed on the error line. They usually display information that can be useful in the process of application development. You may want to hide the system error messages from the users. The WHENEVER statement can be used to prevent system error messages from being displayed. With the help of the WHENEVER statement, you can use the ERROR statement which will display a suitable message to replace a system error message.



EXIT

The EXIT statement is used to terminate the execution of a compound 4GL statement (a block, a loop, etc.) and to pass the program control to the first statement following the END keyword of the corresponding statement. When 4GL encounters the EXIT keyword, the part of the compound statement located between the EXIT keyword and the END keyword is skipped. The following statements can be combined with the EXIT statement:



Element	Description
Exit Code	A 4GL expression that returns a positive integer no greater than 256

The EXIT statement is also used to terminate the program. The EXIT PROGRAM statement is used to terminate the currently running program, the other EXIT statements do not terminate the program. They only transfer the control out of the current program structure.

Exiting a Compound Statement

4GL statements present in the diagram above support the EXIT keyword. The EXIT *statement* keywords transfer the control to the first statement that follows the END *statement* keywords.



To terminate the DISPLAY ARRAY statement, the EXIT DISPLAY keywords are used. To terminate both INPUT and INPUT ARRAY statements, the EXIT INPUT keywords are used.

The EXIT statements can appear only in the corresponding statements. E.g. EXIT WHILE statement can appear only within the WHILE loop, which will transfer the control to the first statement following the END WHILE keywords.

Exiting a Function

You can use the EXIT keyword to leave a function. However, if the EXIT FUNCTION statement is executed, nothing is passed to the calling routine, thus this method cannot be used if you want to return values from a function.

The RETURN keyword performs the function of EXIT statement and returns one or more values to the calling routine. The GO TO and WHENEVER GO TO statements cannot transfer control out of the FUNCTION program block.

Exiting a Report

The EXIT REPORT statement is used to leave the REPORT program block. A report does not return anything to the calling routine, so the RETURN keyword cannot be used within the REPORT program block. However, a report performs output to specified destination. You can also use the FINISH REPORT and TERMINATE REPORT keywords. The GO TO and WHENEVER GO TO statements cannot transfer control out of the REPORT program block.

Exiting a Program

Unlike other EXIT statements, the EXIT PROGRAM statement terminates not a separate program block or compound statement but the program itself. It can occur in any part of the source code and is not linked to a certain statement.

In the example below the EXIT PROGRAM statement is used within the MENU statement.

```
MENU "Main Menu"  
...  
COMMAND "Exit" "Close the program"  
EXIT PROGRAM  
END MENU
```

The other example shows the usage of the EXIT PROGRAM statement in an ON KEY block

```
DISPLAY ARRAY my_arr TO my_scrarr.*  
...  
ON KEY (F20)  
MESSAGE "You now will exit the program"  
SLEEP 1  
EXIT PROGRAM  
END DISPLAY
```



The effect of the EXIT PROGRAM is the same as the effect of the END MAIN statement, both terminate the program.

Exit Code

The EXIT PROGRAM keywords can be followed by the exit code. It can be any whole number less than 256. This exit code value is saved if a program has been run with the help of the RUN statement with the RETURNING clause. The exit code value is returned as an integer that needs 2 bytes for storage; it contains the information on the termination status of the program:

- The low byte specifies the termination status of the program executed with the help of the RUN statement. The value of this status can be calculated by using the formula: integer value modulo 256.
- The high byte contains the low byte from the EXIT PROGRAM statement of the program executed by the RUN statement. Its value can be recovered by dividing the integer value by 256.



FINISH REPORT

The FINISH REPORT statement is used to complete a report. It cannot be used within a REPORT program block.



Element	Description
Report Name	The name of the report used in the REPORT program block

The FINISH REPORT statement specifies the end of a report driver and the termination of report processing. It can be used in the MAIN or FUNCTION program block. The FINISH REPORT *report-name* may occur only after the START REPORT *report-name* statement and at least one OUTPUT TO REPORT *report-name* statement, where the *report-name* is the same.

If a report is defined with the ORDER BY clause and it does not contain the EXTERNAL keyword, or if it specifies aggregates based on all the input records, it is processed twice by 4GL. During the first pass, the database server is used to sort the data which are then stored in a temporary file. During the second pass, the aggregate values are calculated, if any, and the output from temporary files is produced.

The following actions are performed when 4GL encounters FINISH REPORT statement:

- The second pass is completed, if it is a report with two passes. These “second pass” activities handle the calculation and output of any aggregate values that are based on all the input records in the report, such as COUNT(*) or PERCENT(*) with no GROUP qualifier. (rewrite)
- The AFTER GROUP OF blocks are executed
- The PAGE HEADER, ON LAST ROW, and PAGE TRAILER blocks are executed
- The data from output buffers are copied to the destination specified in either START REPORT or OUTPUT section. If no destination is specified, the output is performed to the screen.
- The select cursor on any database table is closed in order to perform input or aggregate calculations
- The memory for the local variables of BYTE or TEXT data types is deallocated
- The report processing is terminated, the temporary files are deleted

The FINISH REPORT statement cannot access temporary tables that belong to different databases. If a different database is opened (by means of the DATABASE statement) while the two-pass report is being processed, 4GL produces an error.



FOR

The FOR statement specifies a loop, the statements within this loop are executed a definite number of times.



Element	Description
Counter	A variable of INTEGER or SMALLINT data type that serves as an index for the statement block.
TO Clause	A clause where the upper and lower limits for the counter are specified
STEP Clause	An optional clause specifying the size of one step
Logical Block	A block that must include at least one executable statement. It can also include the definition block

The statements in the Logical Block are executed the specified number of times, unless the execution of the FOR statement is terminated by the EXIT FOR statement. The FOR statement is typically used, when you know the upper limit of the iterations required. If you are not sure how many times the loop is to be repeated, it is advisable that you use the WHILE loop instead.

The TO Clause

The TO clause contains the *counter* and the minimum (*first*) and maximum (*last*) value of this counter.



Element	Description
First	An integer expression which specifies the initial value of the counter
Last	An integer expression which specifies the upper limit for the counter value

The last value corresponds to the maximum number of loop iterations by default, unless the step is other than 1.

The value of the *Counter* changes each time when 4GL passes through the FOR loop. When 4GL passes through the loop for the first time, the counter is set to the value of the *first* expression. Each time when 4GL passes through the loop again, the value of the counter is increased by the step value in the STEP clause. By default the step value is 1. This value is added to the *counter* each time the loop is iterated.

When the value of *counter* becomes equal to the maximum number of iterations specified to the right of the TO keyword, the FOR loop is terminated. That is, the FOR loop is terminated when the Boolean expression "first=last" is TRUE. The control is transferred to the first statement that follows the END FOR keywords.

```
FOR count = 1 TO 5
```



```

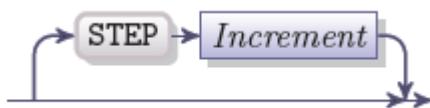
IF state[i] = status_name THEN
    RETURN i
END IF
END FOR

```

The last expression specifying the maximum number of iterations cannot be a smaller number than the first expression, if the STEP *increment* is a positive value. If either of these expressions returns NULL, the loop cannot be terminated, because the Boolean expression cannot be evaluated as TRUE.

The STEP Clause

The STEP clause is used to define the number (increment) by which the counter will be increased each time a FOR loop is iterated.



Element	Description
Increment	An integer expression whose value is added to counter after each iteration of the statement block.

In the example below only odd records from the program array are displayed to the screen array, because the *increment* is 2.

```

FOR iter = 1 TO 10 STEP 2
    DISPLAY my_arr[iter] TO my_screc[i].*
    LET i=i+1
END FOR

```

If the increment is specified as a negative value, the last expression to the right of the TO keyword must be smaller than the first.

Before executing the FOR loop, 4GL compares the counter value to the last value. If the counter is larger than the maximum number of iterations, the FOR loop is not executed and the control is transferred to the statements that follow this loop.

The Logical Block

The logical block which can be included into the FOR statement has the following structure:



Element	Description
---------	-------------



Declaration Clause	The optional DEFINE statement
Statement List	Executable statements or special keywords

The Logical Block can include the DEFINE statement used to declare spot variables. These variables are visible only within the FOR statement and cannot be referenced from outside the FOR loop. For more information about the [DEFINE](#) statement see the corresponding section of this reference. The DEFINE statement of the Logical Block is optional and can be omitted.

The Statement list can include the following:

- Executable 4GL statements
- SQL statements that can be embedded in the 4GL code
- EXIT FOR keywords
- CONTINUE FOR keywords

The CONTINUE FOR Keywords

When 4GL encounters the CONTINUE FOR statement within a FOR loop, it interrupts the current iteration and begins the next iteration of the loop. The CONTINUE FOR has the following effects:

- All the statements between the CONTINUE FOR and END FOR statements are skipped
- 4GL starts the new iteration provided that the *counter* value is less than the maximal number of iterations. (Otherwise the statements that follow END FOR keywords are executed instead)

The EXIT FOR Keywords

The EXIT FOR statement terminates the FOR loop. When 4GL encounters this statement, it skips all the statements that follow the EXIT keywords and starts execution of the first statement following the END FOR keywords.

The END FOR Keywords

The END FOR statement specifies the end of the FOR loop. When 4GL encounters this statement, it compares the *counter* with the maximum number of iterations specified in the TO clause. If the *counter* value is less than the maximum number of iterations, 4GL executes another iteration of the loop. If the value of the *counter* is greater than or is equal to the maximum number of iterations, 4GL leaves the FOR loop and executes the statements that are following the END FOR keywords.

The FOR Loop in Databases with Transactions

If you use one or more SQL statements within a FOR loop that refers to a database with transaction logging, the whole FOR loop should be within a transaction. Otherwise one of the following problems might occur:

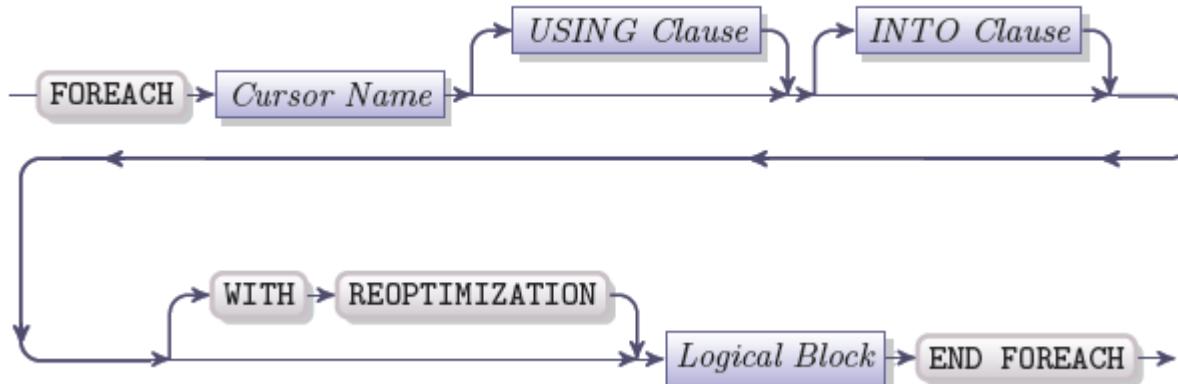
- Difficulties in determining the extent to which the database integrity has been compromised
- In case a database is corrupted during the execution of a FOR loop, it might be hard to restore the condition of the database in which it has been before the execution of the loop

It is also advisable that you use the FOREACH and WHILE statements within the transactions, if they contain SQL statements.



FOREACH

The FOREACH loop is used to apply some actions to each row of data that a cursor returns from query.



Element	Description
Cursor Name	The name of a previously declared cursor
USING Clause	Used to specify a variable or variables used as the query criteria
INTO Clause	Used to specify variable or variables for storage of the query results
Logical Block	A block that must include at least one executable statement. It can also include the definition block

The FOREACH statement combines the features of the OPEN, FETCH and CLOSE statements. It retrieves and processes the rows of data that have been selected from a database. The FOREACH statement opens the specified cursor, fetches the rows selected one by one and closes the cursor after it has processed the last row.

A cursor must be declared by means of the DECLARE statement, before a FOREACH statement for this cursor can appear in the source code. If you try to FETCH a cursor that has not been declared, 4GL produces a compile-time error. Any type of cursor (SCROLL cursor, cursor FOR UPDATE or cursor WITH HOLD) can be used with the FOREACH statement, but it processes the rows one by one only in sequential order.

The FOREACH statement closes cursor when the last of the selected rows has been fetched, or if the WHENEVER NOTFOUND condition is true (if status built-in variable is equal to 100).

A Cursor in the FOREACH loop

The cursor name must follow the FOREACH keyword. This cursor must be declared above the FOREACH loop in the same program module. If there is no cursor identifier following the FOREACH statement, a runtime error will occur.

In the example below the cursor *c_contact* is declared before the FOREACH statement, which assigns the values of each row selected from the table *contact* by the SELECT statement to elements of a program array *cont_arr*.



```
DECLARE c_contact CURSOR FOR
    SELECT * FROM contact
    WHERE contact.cont_ord = p_company_id
FOREACH c_contact INTO cont_arr[i].*
LET i = i+1
END FOREACH
```

The FOREACH statement executes the OPEN statement implicitly, and then it executes the FETCH statement, which normally exits when NOTFOUND is returned, and after that the CLOSE statement is executed. If the FOREACH statement includes the WHENEVER ERROR GOTO statement or WHENEVER ERROR CALL statement and the error returned by the implicit FETCH statement is other than NOTFOUND, the CLOSE statement is not executed. This means that the cursor will not be closed though the program control has been transferred out of the FOREACH loop and it needs to be closed explicitly by means of the CLOSE statement.

The FOREACH statement does not automatically reflect the number of rows that have been fetched in the sqlca.sqlerrd[3] element of the built-in array. To retrieve the number of rows fetched, you must maintain your own row counter.

The INTO Clause

The INTO clause specifies one or more variables separated by commas that are used to store the values from the rows processed by the FOREACH statement.



Element	Description
Variable List	A list of one or more variables separated by commas

The variables must match the columns of the specified table in number, order and data types. The INTO clause in the example above stores the values retrieved from the database table into a program array:

```
FOREACH c_contact INTO cont_arr[i].*
```

The INTO clause is optional, it may not be present, if the INTO clause is already present in the SELECT statement for which the cursor used in the FOREACH statement has been declared. However, it is advisable that you include the INTO clause into the SELECT statement only if the rows are **not** retrieved into a program array, otherwise, use the INTO clause of the FOREACH statement. Below is an example of the INTO clause within the SELECT statement instead of the FOREACH statement.

```
DECLARE c_contact CURSOR FOR
    SELECT company_name INTO comp_name FROM contact
    WHERE contact.cont_ord = p_company_id
FOREACH c_contact
```



END FOREACH

	Note: You cannot have the INTO clause both in the SELECT statement referring to the cursor and in the FOREACH statement.
--	---

The USING clause

The USING clause specifies one or more variables separated by commas which values will provide the search criteria for the query performed. This clause should be included into the FOREACH statement, if the search criteria are provided by the user - that is if the processed cursor is associated with a prepared statement containing placeholders.



Element	Description
Variable List	A list of one or more variables separated by commas

If the cursor is associated with a statement without placeholders (?), the USING clause must not be used. The number of variables in the variable list and their data types must correspond to the number and positions of the placeholders in the prepared statement associated with the cursor. For more information about the placeholders see the [PREPARE](#) statement.

In the example below, the cursor has been declared for a SELECT statement without the WHERE clause, the search criteria are provided by the USING clause of the FOREACH statement:

```
PREPARE prepared_stmt FROM "SELECT * FROM company WHERE city = ?"  
DECLARE c_company CURSOR FOR prepared_stmt  
PROMPT "Enter the city : " FOR city_var  
FOREACH c_company USING city_var INTO comp_arr[i].*  
LET i = i+1  
END FOREACH
```

The number and data types of the variables used in the USING clause must correspond to the number and data types of the placeholders used in the prepared SELECT statement. If both USING and INTO clauses are present in a FOREACH statement, the INTO clause must follow the USING clause.

The WITH REOPTIMIZATION Keywords

The WITH REOPTIMIZATION keywords enable you to re-optimize your query design without having to rewrite the query. When you prepare a SELECT or EXECUTE PROCEDURE statement, the database server uses a query-design plan to optimize the performance of the query. If the data values associated with these statements are modified later, the query may become inefficient. To prevent such situation you can:



- Prepare the SELECT or EXECUTE PROCEDURE statement again
- Use the FOREACH statement with the WITH REOPTIMIZATION keywords

The reoptimization does not rebuild all the statements, it just rebuilds the query plan, thus it makes new values applicable to the query plan. This process takes less time and requires fewer system resources than re-preparing.

The Logical Block

The logical block which can be included into the FOR statement has the following structure:



Element	Description
Declaration Clause	The optional DEFINE statement
Statement List	Executable statements or special keywords

The Logical Block can include the DEFINE statement used to declare spot variables. These variables are visible only within the FOREACH statement and cannot be referenced from outside the FOREACH loop. For more information about the [DEFINE](#) statement, see the corresponding section of this reference. The DEFINE statement of the Logical Block is optional and can be omitted.

The Statement list can include the following:

- Executable 4GL statements
- SQL statements that can be embedded in the 4GL code
- EXIT FOREACH keywords
- CONTINUE FOREACH keywords

The CONTINUE FOREACH Keywords

The CONTINUE FOREACH keywords terminate the processing of the current row within a FOREACH loop, 4GL comes back to the first statement of the FOREACH loop, fetches the next row and processes it.

```
DECLARE c_cont CURSOR FOR
    SELECT company_name, company_id, FROM contact
    WHERE contact.cont_loc = p_company_location
FOREACH c_cont INTO cont_arr[i].company_name, cont_arr[i].company_id
LET i = i+1
    IF company_id < 100 AND company_id > 10 THEN
```



```
CONTINUE FOREACH
END IF
END FOREACH
```

The EXIT FOREACH Keywords

The EXIT FOREACH statement terminates the execution of the FOREACH loop and 4GL starts executing the first statement that follows the END FOREACH keywords. All the statements between the EXIT FOREACH and END FOREACH keywords are ignored.

```
FOREACH c_contact INTO cont_arr[i].*
LET i = i+1
IF i > 50 THEN
EXIT FOREACH
END IF
END FOREACH
```

The END FOREACH Keywords

The END FOREACH keywords specify the end of the FOREACH loop. When 4GL encounters the END FOREACH keywords, it returns to the beginning of the loop and tries fetching another row. The FOREACH loop is iterated until there remain no more rows returned by the query. When there are no rows left, 4GL starts executing the first statement that follows the END FOREACH keywords.

The FOREACH Loop

The FOREACH statement specifies a loop which is iterated each time a new row is fetched and processed by the FOREACH statement. If the cursor returns no rows, the statements within the FOREACH loop are skipped and 4GL starts executing the statements that follow the END FOREACH keywords.

The FOREACH Loop in Databases with Transactions

If you use one or more SQL statements within a FOREACH loop that refers to a database that has transaction logging, the whole FOREACH loop should be placed within a transaction. Otherwise there might occur one of the following problems:

- Difficulties in determining the extent to which the database integrity has been compromised
- In case a database is corrupted during the execution of a FOREACH loop, it might be hard to restore the condition of the database in which it has been before the execution of the loop

It is advisable that you also use the FOR and WHILE statements within the transactions, if they contain SQL statements.

If the cursor has been declared FOR UPDATE and it is **not** a WITH HOLD cursor, the FOREACH loop must be placed within a transaction. A FOR UPDATE **and** WITH HOLD cursor can be opened outside the transaction, however it will not be possible to rollback the changes made with the help of such cursor. In this situation each UPDATE WHERE CURRENT OF will be committed as a singleton transaction.



FREE

The FREE statement is an SQL statement, that can be directly embedded into the 4GL source code. It is used to release resources allocated to a cursor, a prepared statement or a variable of a large data type (TEXT or BYTE, in particular).

The FREE Statement for Cursor

If used to release resources allocated to a cursor, the FREE statement has the following syntax:

```
FREE cursor_name
```

The total number of the open cursors, that are allowed for a single process at a time, is limited by an amount of the system memory available.

Once freed, the cursor cannot be used by any other statement. To use a closed cursor again, you should reopen it: declare the cursor once more within the source code. Note, that the cursor which has not been explicitly closed cannot be freed.

Releasing Resources Allocated to a Cursor Declared with a CURSOR Variable

To free the resources allocated to a cursor, declared with the help of a CURSOR variable, the `Free()` method should be called. It takes no arguments and returns a value of `sqlca.sqlcode` on its invocation:

```
CALL cur.Free()
```

After the statement above has been executed, the cursor cannot be opened and used again within the application.

The `Free()` method is implicitly invoked when all the references to the cursor object are closed.

The FREE Statement for Prepared Statement

The FREE statement followed by the name of a prepared statement releases the resources that the program has allocated to this prepared statement:

```
FREE prepared_stmt_id
```

After the FREE statement has been executed, the statement identifier specified in it cannot be referenced by the EXECUTE and the DECLARE cursor statements, until the same or another statement is linked to this identifier once again within an application source code.

If the FREE statement is applied to a prepared statement which is not associated with a cursor, it frees the resources in both a database server and an application development tool. If a prepared statement is associated with a cursor, the structure like `FREE stmt_id` can releases only the resources of an application development tool. The database server resources can be freed only under condition that the cursor is freed.



Releasing Resources Allocated to a Statement Declared with a PREPARED Variable

To free the resources associated with a prepared statement by defining a PREPARED variable, the `Free()` method is typically used. It takes no arguments and returns a value of `sqlca.sqlcode` on its invocation:

```
CALL variable_name.Free()
```

After the `Free()` method has been executed, the statement you prepared with a variable of the PREPARED data type cannot be referenced again without preparing it repeatedly by means of the `Prepare()` method.

The FREE Statement for Large Objects

The FREE statement followed by the name of a variable of the TEXT or BYTE data type releases resources allocated for the specified variable data storage:

```
FREE text/byte_var_name
```

Depending on the place of the TEXT or BYTE variable location, the runtime system may release the memory or delete a file intended to store the data.

The resources for the variables of a local scope are freed automatically on returning from the MAIN or function blocks during an application execution.

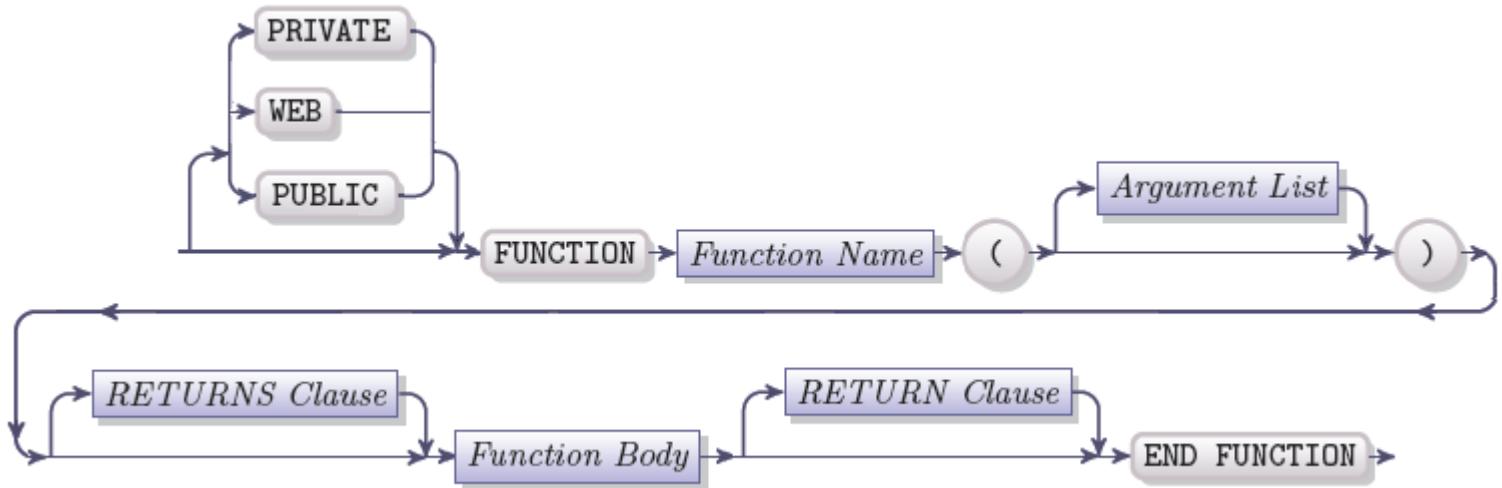
After the FREE statement has been declared, the specified TEXT or BYTE variable cannot be used again until it is reconfigured using the LOCATE statement.

	Note: On a program termination, all the temporary files containing large data objects are deleted without notice.
--	--



FUNCTION

The FUNCTION statement indicates the beginning of the FUNCTION program block. The FUNCTION program block has the following structure:



Element	Description
Function Name	The name that you declare for this 4GL function.
Argument List	A list of zero or more formal arguments of the function separated by commas
RETURNS clause	An optional clause used to make the function compliant with the web services
Function Body	Executable statements that function contains
RETURN Clause	Optional clause that returns values from the function into the calling routine

The FUNCTION program block is a set of statements that has a unique name. A specified set of statements can be called from any part of the source code in any module within one program.

The FUNCTION statement:

- declares the name of the function and its formal arguments
- defines the FUNCTION program block

The FUNCTION block is a separate program block like the MAIN or REPORT program blocks, therefore it cannot occur within another program block, e.g. it cannot be a part of the MAIN program block. It cannot also appear within another FUNCTION block.

A FUNCTION block is the program block delimited with the help of the FUNCTION keyword at the beginning and END FUNCTION keywords at the end.

If a function returns a single value, it can be invoked within a 4GL expression as its operand (see the "Functions used as Operands in 4GL Expressions" section for more details). If a function returns more than one value, it must be called with the help of the CALL statement. If a function returns some values, it should contain the RETURN clause and the calling routine should have the RECEIVING clause. The variables in the



RETURN clause of a FUNCTION must match the variables in the RETURNING clause of the CALL statement in order, number and data types.

Name of a Function

The name of a function must be unique within one application. This means that no other function and no report can have the same name. It cannot also be the same as the name of a built-in 4GL function. The name of the function is not case sensitive.

Formal Arguments of a Function

The formal arguments of a function are the identifiers enclosed in parentheses that follow the function name. They will be transferred to the function, when it starts executing; they are referenced by value and must match the formal arguments in the CALL statement in order, number, they must be of compatible data types. The names of the arguments must be unique within the function. They are local to the function and cannot be referenced outside of it.

```
FUNCTION my_function (p_lname, p_fname)
```

	Note: If a function requires no formal arguments, there still must be an empty argument list enclosed in parentheses: <i>function()</i>
--	--

Function Body

A function body consists of all the statements between the FUNCTION keyword at the beginning and END FUNCTION keywords at the end (or the RETURN clause, if it is present). The statements within the function body are executed when the function is invoked.



Element	Description
Declaration Clause	The optional DEFINE statement
Statement List	Executable statements

An empty function without the Logical block can be defined, though such function does not perform any actions and thus the function body is represented as obligatory in the syntax diagram:

```
FUNCTION test_function()  
END FUNCTION
```

However, an empty function allows you to test the other parts of the program before the FUNCTION program block is written.



The Declaration Clause

If a function has one or more formal arguments, the data types of these arguments must be declared in the DEFINE block. The DEFINE block must precede any executable statements of the FUNCTION block.

The actual arguments in the CALL statement may not be of the same data types as the formal arguments of the function, provided that they are of compatible data types. If the data type conversion is impossible, you will receive a runtime error (see "Conversion of Data Types" section for more information).

Function Local Variables

Other variables used in the FUNCTION program block must also be declared in the DEFINE clause. These variables are local to the function and are not visible to other program blocks; their names must be unique within a function.

Global and module variables can be referenced in the FUNCTION block, however if the name of a global or module variable matches the name of a local variable of a function, the FUNCTION program block will use the local variable.

You can declare a variable LIKE a column in a database, but to do so you must have a default database declared for the given program module (see the "DATABASE" statement section for more information). The DATABASE statement can occur within a FUNCTION block to specify a new current database.

The GOTO and WHENEVER ... GOTO statement can transfer control only within the same FUNCTION program block.

Executable Statements

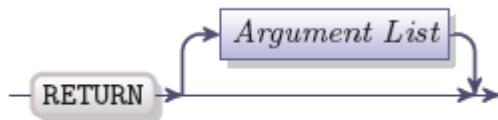
A function can include any executable statement which will be executed when the function is invoked.

```
FUNCTION cont_info (p_cust_id)
  DEFINE
    p_cust_id INT,
    l_cust_lname CHAR(30)
  SELECT customer_lname INTO l_cust_lname FROM customer_info
    WHERE customer_id = r_cust_id
  RETURN l_cust_lname
END FUNCTION
```

The above example is a FUNCTION program block which has one formal argument, the DEFINE block and one executable statement: the SELECT statement. It also contains the keywords that indicate the beginning and the end of the block and the RETURN clause which returns the value of the *l_cust_lname* variable back to the calling routine.

The RETURN Clause

The RETURN clause is optional and must be included into a FUNCTION program block only if a function returns some values to the calling routine.



Element	Description
Argument List	The list of values returned by the function

The values specified in the RETURN clause must match the values specified in the RETURNING clause of the CALL statement in order and number, they must also be of compatible data types. If the RETURN clause contains no arguments, it transfers program control back to the calling routine without returning any values.

A programmer defined or a built-in function that does not have the same name as a built-in operator, and that returns a single value, can be used as an operand in a 4GL expression.

```
MAIN
...
IF func2() < 0 THEN
    EXIT PROGRAM
END IF
...
END MAIN
FUNCTION func2()
DEFINE var1 INT
PROMPT "Enter your number " FOR var1
RETURN var1
END FUNCTION
```

A function calling used as an operand can include formal arguments, if they are required. For more information about the RETURN statement see the "RETURN" section of this chapter.

Public and Private Functions

4GL functions are public by default if no other option is specified. Web functions can be public only and cannot be made private. All other functions can be made private or explicitly public using the PRIVATE or PUBLIC keywords.

To make a function public you should either use the FUNCTION keyword without any preceding keywords or with the PUBLIC keyword before it. A public function can be called from other modules included into the program. It is also visible, if the module is imported using the [IMPORT](#) statement or if the module is a part of the linked 4GL library. Here is an example where both functions are public:

```
FUNCTION func1()
```



```
    ...
FUNCTION
PUBLIC FUNCTION func2()
    ...
END FUNCTION
```

To make a function private, you should use the PRIVATE keyword before the FUNCTION keyword. The private functions visibility is limited only to the module in which they are declared. They cannot be called from other modules of the same program, if they are present in an imported module or in a linked library, they are inaccessible by the main program.

Web Services Compliant Functions

To make a 4GL function usable with the web services, you should add the following in the function declaration part:

1. Add the WEB keyword before the FUNCTION statement.
2. Add the RETURNS clause right after the formal argument list.

Web functions are public by default and cannot be made private.

Any 4GL function can be converted to a web function in such a way, e.g.:

```
#a 4GL function
FUNCTION my_func(c)
...
RETURN a,b
END FUNCTION

#a web 4GL function
WEB FUNCTION my_func(c) RETURNS(a,b)
...
RETURN a,b
END FUNCTION
```

The RETURNS Clause

The RETURNS keyword is not an independent statement. It should not be confused with the RETURN statement which is a part of the RETURN clause or with the RETURNING clause of the CALL statement, though they all deal with the values returned from the function.

The RETURNS clause is used to declare the return type(s) of a function. It also causes the 4GL function to enforce the values returned by the function if they do not match the function declaration. The RETURNS clause is obligatory for the web functions. However, its usage is not restricted to web functions only and it can be used in any 4GL function where it is not obligatory.

The RETURNS clause has the following syntax:



Element	Description
Argument List	The list of values returned by the function

The RETURNS and RETURN clauses can be used within the same 4GL function, only one of these clauses can be present at a time or they may be absent altogether. The RETURNS clause is always present in a web function.

The values are returned from the function by means of these two clauses using such rules:

- When a function is declared with RETURNS clause does not have the RETURN clause, it will implicitly return the stated values when the function completes.
- When a function has both the RETURNS and RETURN clause, the values specified in the RETURN statement will be returned by the function
- The RETURNS clause will cause the compiler to enforce the number of values returned by a function, where the user has supplied a mis-matching number of values in a RETURN statement:
 - The function returns all values listed in the RETURNS clause, if the RETURN statement does not contain any values or contains fewer than the RETURNS clause.
 - If the RETURN clause contains more values than the RETURNS clause does, a compile-time error occurs.

For more information about the web services see the "Web Services" guide.

The END FUNCTION Keywords

The END FUNCTION keywords indicate the end of the FUNCTION program block. They can be followed by another FUNCTION program block or a REPORT program block.



GLOBALS

The GLOBALS statement is used to declare global variables, which can be referenced by any module or program block within a program.



Element	Description
Declaration Clause	The DEFINE statement where the global variables are declared

There are four types of variables in Querix 4GL:

Variable Type	Declare Location	Scope of Reference
Local	Declared within a program block (MAIN, FUNCTION or REPORT)	The block where it is declared
Module	Declared before any program block at the beginning of a program module	The module where they are declared
Global	Declared with the help of the GLOBALS statement	Any program module where the GLOBALS file is referenced
Built-in	They do not require declaration	The entire 4GL application

To declare a global variable, follow these steps:

- Declare variables with the help of the GLOBALS statement in a separate file, that does not contain any program block other than the GLOBALS block
- The DATABASE statement can precede the GLOBALS keyword in the GLOBALS file, if it is required
- Specify the GLOBALS "filename" before any program block in modules, that reference global variables
- The GLOBALS file must be compiled and linked to the other files of the program

It is possible to declare global variables in a program module that contains other program blocks besides the GLOBALS program block. However, the scope of reverence of such variables will be only the module where they are declared. The file that contains other program blocks besides the GLOBALS cannot be referenced by the GLOBALS "filename" statement, because it includes executable statements.

The number of files that contain global variables is not limited, thus one program can have several GLOBALS files. More than one file containing global variables can be referenced with the help of the GLOBALS "filename" statement within one module.

Global Variables before the MAIN Program Block

To declare global variables you must use the GLOBALS statement in a separate file or before any program block. The GLOBALS statement must appear before any other program block (that is before MAIN, FUNCTION and REPORT blocks), the variables declared in such a way will have the modular scope of reference. The GLOBALS program block must contain at least one DEFINE statement. The END GLOBALS statement must follow the variables declaration; they are used to mark the end of the GLOBALS block.



If global variables are declared with the help of the LIKE keyword, the DATABASE statement must precede the GLOBALS keyword.

Here is a fragment of a program with global variables declared in a program module which contains MAIN and FUNCTION block:

```
DATABASE cms
GLOBALS
    DEFINE g_language_id LIKE qxt_language.language_id,
        g_language_name LIKE qxt_language.language_name
END GLOBALS
MAIN
...
CALL get_l_name()
...
MAIN
FUNCTION get_l_name()
SELECT language_name INTO g_language_name FROM qxt_language
    WHERE language_id = g_language_id
DISPLAY g_language_name TO f_lang_name
END FUNCTION
```

The variables *g_language_id* and *g_language_name* declared in the GLOBALS block are referenced by the function without being passed to it as formal arguments.

This file cannot be referenced with the help of the GLOBALS "filename" statement, because it contains other program blocks besides the GLOBALS (i.e. the MAIN and FUNCTION program blocks). The variables declared with the help of the GLOBALS statement in a file that contains other program blocks have the same features as module variables; their scope of reference is only the module where they are declared. To extend the scope of their reference the variables need to be declared in a separate file (called the GLOBALS file) and then referenced in the modules in which they will be used with the help of the GLOBALS "filename" statement.

You can declare local variables with the same names. The local variables will be used in the program blocks where they are declared rather than global ones, if their names match.

	Note: You cannot declare a module variable with the same name as a global variable declared in the same module in the GLOBALS block and not in a separate GLOBALS file, otherwise you will receive a compile-time error.
--	---

Though you can have several GLOBALS statement in one program, the names of all the global variables must be unique for the same scope of reference. You cannot declare two global variables with the same name even if they occur in different GLOBALS statements. It may result in a compile-time error or in unpredictable runtime behaviour.



Global Variables in the GLOBALS File

A GLOBALS file is a file that can contain only GLOBALS, DEFINE and DATABASE statements. It cannot contain executable statements. The GLOBALS file is used to extend the scope of reference of the variables declared as the global variables to all the program modules which contain the GLOBALS "filename" reference, where the "filename" is the name of the GLOBALS file.

To create and use the GLOBALS file, do the following:

1. Create a 4GL file which contains:
 - o The DATABASE statement (optional). It is required if any of the global variables is declared LIKE a database column. It can also be included if you want to use a default database. The DATABASE statement in a GLOBALS file will make the database specified in it the default database for each program module where the GLOBALS file is referenced. The DATABASE statement must precede the GLOBALS statement
 - o The GLOBALS statement
 - o One or more DEFINE statements used to define the global variables
 - o The END GLOBALS keywords
2. Use the GLOBALS "filename" statement in a program module that uses the global variables. It must precede the first program block. The *filename* is the name of the GLOBALS file together with extention. The END GLOBALS keywords are not required.

Below is the example of a GLOBALS file called *cms_globals*:

```
DATABASE cms
GLOBALS
    DEFINE g_language_id LIKE qxt_language.language_id,
        g_language_name LIKE qxt_language.language_name
END GLOBALS
```

No more modules can occur after the END GLOBALS statement. The GLOBALS file can then be referenced in the program modules as follows:

```
GLOBALS "cms_globals.4gl"
MAIN
...
MAIN
FUNCTION get_l_name()
SELECT language_name INTO g_language_name FROM qxt_language
    WHERE language_id = g_language_id
DISPLAY g_language_name TO f_lang_name
END FUNCTION
```



	Note: The GLOBALS "filename" statement can be followed by any number of program modules, but it must occur at the beginning of every module that uses the corresponding global variables. It must also be located before the first program block.
---	--

You can reference two or more files that contain global variables in one module:

```
GLOBALS "cms_globals.4gl"  
GLOBALS "additional_cms_globals.4gl"
```

The GLOBALS "filename" statement can appear in any program module including those blocks which do not contain the MAIN program block.

If a local variable has the same name as a global variable, the local variable rather than the global one will be visible in the scope of reference of this local variable. Likewise the module variable with the same name will be visible in the corresponding module rather than the global one.

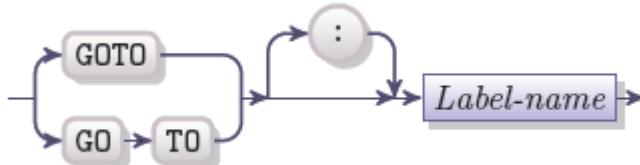
Querix 4GL does not perform the check for the conflict between the names of the global variables and the built-in function calls, so it is better to avoid using the names of the built-in functions (i.e. infield(), arr_curr(), etc.) as the names of the global variables.

Querix 4GL performs check for the conflict between the names of the built-in variables and module or global variables. If you try to declare a global or a module variable with the same name as a built-in variable (i.e. status, int_flag, quit_flag, etc.), a compile time error will occur. However, Querix 4GL does not check for the conflict between the names of the built-in variables and local variables.



GOTO

The GOTO statement is used to transfer program control within the range of the same program block. It transfers the control to the place where the corresponding LABEL statement is located.



Element	Description
Label-name	The name of a previously declared label

When 4GL encounters the GOTO statement, it transfers control to the LABEL statement, where the *identifiers* are the same. It begins executing the first statement that follows the LABEL statement skipping all the statements that follow the GOTO statement. The corresponding LABEL statement can be located anywhere in the same program block, either above the GOTO statement or below it. See also the section about the "[LABEL](#)" statement.

The GOTO statement can transfer program control to the LABEL statement only if:

- The GOTO and the LABEL statement have the same *label-name*
- The GOTO and LABEL statement occur within the same program block (MAIN, FUNCTION or REPORT)

It is not advisable that you overuse the GOTO and LABEL statements, it may lead to difficulties in reading and maintaining the code and in infinite loops. There are alternative methods of transferring program control, they are:

- CASE, FOR, IF and WHILE statements with Boolean expressions
- CALL, OUTPUT TO REPORT and WHENEVER statements
- EXIT keyword with the corresponding statement which terminates this statement (see the section on the "EXIT" keyword)
- The CONTINUE keyword with the corresponding statement (see the section of the "CONTINUE" statement)

The GOTO statement can prove useful in the situations when you need to exit from a deeply nested loop:

```
LABEL retry:  
...  
FOR i = 1 TO 20  
    FOR a = 5 TO 15  
        ...  
        IF ent_val < 0 THEN  
            GOTO :retry
```



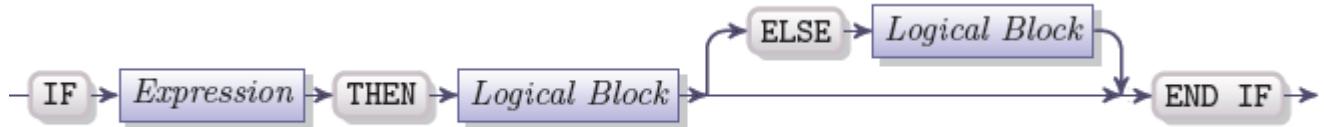
```
    ELSE
    ...
END IF
END FOR
END FOR
```

You can also place a colon before the *label-name* in the GOTO statement to indicate that it conforms to the ANSI standard.



IF

The IF statement executes statements depending on whether the desired condition is met. It can contain only one block of statements which will be executed only if the condition specified in the statement is met. It can also have two blocks of statements one of which is executed when the condition is met and the other is executed when it is not met.



Element	Description
Expression	A Boolean expression
Logical Block	A block that contains variables declaration and executable statements

The IF statement is used with a Boolean expression. If the Boolean expression specified after the IF keyword is evaluated as TRUE, the statements between the THEN keyword and the ELSE keyword (or between the THEN keyword and the END IF keywords, if the ELSE clause is omitted) are executed. After than the statements that follow the END IF keywords are executed.

If the Boolean expression is evaluated to FALSE, 4GL executes the statements after the ELSE keyword (if it is present). If there is no ELSE block in the IF statement, 4GL skips the statements within the IF statement entirely and starts executing the statements that follow the END IF keyword. The Boolean expression also returns FALSE, if it is evaluated as NULL (unless it is used with the IS NULL operator).

The IF loops can be nested in one another, the limit is about 20 loops, the limit also depends on the number of the FOR and WHILE loops used in the module. If you want to use several IF loops to test the same value against different results, it is advisable that you use the CASE statement instead.

```
LABEL retry:  
  IF yes_no MATCHES [Yy]  
    CALL tot_inf()  
  ELSE IF yes_no MATCHES [Nn]  
    MESSAGE "The program will be closed in 1 sec."  
    SLEEP 1  
    EXIT PROGRAM  
  ELSE  
    MESSAGE "You should enter either Yy for yes or Nn for no."  
    GOTO :retry  
  END IF  
END IF
```



The Logical Block

The logical block which is included into the IF statement has the following structure:



Element	Description
<i>Declaration Clause</i>	The optional DEFINE statement
<i>Statement List</i>	Executable 4GL or SQL statements

The IF statement can include one logical block, if the ELSE clause is absent or two logical blocks, if it is present. Each of them can have its own declaration clause with the DEFINE statement used to declare spot variables. These variables are visible only within their logical blocks and cannot be referenced from outside the IF statement. The variables declared in the ELSE clause cannot be used outside the ELSE clause. For more information about the [DEFINE](#) statement see the corresponding section of this reference. The DEFINE statement of the Logical Block is optional and can be omitted.

The Statement list can include any executable statements which may use the corresponding spot variables.

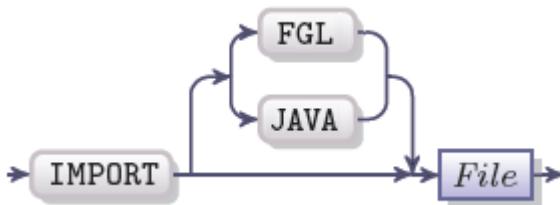


IMPORT

The IMPORT statement can be used to import precompiled resources into a 4GL program. These resources can be of different types. You can import the following modules:

- C Extension libraries
- Java classes
- Individual compiled 4GL files and compiled 4GL libraries

Here is the syntax of the statement:



Element	Description
File	An identifier which references a valid file name without path or extension

The IMPORT JAVA statement used for importing Java modules into a 4GL program is described in more details in the "Lycia II Java Interface" Guide. You can download it from [Querix website](#). This chapter will discuss the importing of C and 4GL modules in more detail.

The IMPORT statement must occur at the beginning of a source file before the MAIN or a FUNCTION section, as well as before all the DEFINE and GLOBALS statements. You should reference one module per IMPORT statement. If you want to import several modules, use several IMPORT statements.

The IMPORT statement then allows to call the functions from the imported module in the 4GL module into which the import was performed without linking them explicitly to the program.

The File Name

For importing any type of module the file name rules are the same. These rules are the same as the rules for any 4GL identifier:

- The name must not be enclosed in quotation marks .
- It must not have a path or an extension attached to it.
- The file name must not contain whitespaces
- It can contain underscores and other printable symbols, provided that the first symbol of the file name is a letter.

Normally in 4GL if a file is referenced by a program, its name path and extension are needed and they are enclosed in quotes. However, for the IMPORT statement the file name also serves as an identifier which can be later used in the code, thus the above mentioned restrictions must take place.

Here are some examples:



```
IMPORT JAVA java.util.regex.Pattern -- imports a Java class
IMPORT FGL my_lib -- imports a 4GL library
IMPORT c_lib -- imports a C library
MAIN
...
END MAIN
```



Note: The name of the module is case sensitive. It is recommended to use lower-case names for C and 4GL libraries.

Importing C Libraries

The imported C library will be used by the 4GL program at compile time and at runtime. For the imported library to work, it must exist as a shared library with .dll extension for Windows systems and .so extension for UNIX/Linux systems and be loadable (environment variables must be set properly).

Once a C library is imported, you do not need to link it to a program during compilation, the Lycia runtime loads the imported libraries dynamically.

Lycia looks for the C library referenced by the IMPORT statement first in the current directory and then in the directory specified by the FGLLDPATH environment variable, if it is set.

The C functions in the imported library that are referenced by the 4GL program must be written with the corresponding conventions to be usable in 4GL code. For more information about C API in Lycia see "Lycia II Developers Guide" the "Using C API in Lycia" chapter.

For importing a C library the library name should immediately follow the IMPORT keyword, because this is the default import mode. Here is an example of a C library imported into a 4GL program.

```
IMPORT c_lib
MAIN
    CALL c_func("Hello World") -- function defined in c_lib
END MAIN
```

Importing 4GL Modules

You can import a compiled 4GL object file or a compiled 4GL library into a program. A compiled object file (.4o) or a compiled 4GL library (.4a) should be referenced without their extensions following the IMPORT 4GL statement. The 4GL modules mentioned in the IMPORT statement are first looked for in the current directory and then in the directory specified by the FGLLDPATH environment variable.

You cannot include circular imports where two or more files import one another, since for one file to compile the other must be already compiled.



If you import a 4GL module in such a way, you do not need to link it to the program using the qlink tool. All the functions from within the imported module are available in the source module it is imported into and can be called like any other 4GL functions.

The elements of the imported module that can be referenced:

- Public functions
- Public constants
- Public user types
- Public module variables

By default all the elements listed above are private. To make them public you should use the PUBLIC keyword when declaring them. For more information see the corresponding [FUNCTION](#), [DEFINE](#), [CONSTANT](#) statements.

Referencing the imported modules

The functions from the imported 4GL modules can be called like normal 4GL functions either by CALL statement or used as operands of 4GL expressions. If a function name is unique, it can be called without any prefixes:

```
IMPORT fgl_lib
MAIN
    CALL myfunction() -- function defined in fgl_lib
END MAIN
```

If you imported several modules and some functions in them have the same names, you can distinguish between the functions by adding the prefix which is the name of the imported module before the called function name:

```
IMPORT fgl_lib
IMPORT my_lib
MAIN
    CALL fgl_lib.myfunction() -- function defined in fgl_lib
    CALL my_lib.myfunction() -- function defined in my_lib
END MAIN
```

If a function in the imported module has the same name as a function defined in the current module, call to the function defined in the imported file must have a prefix:

```
IMPORT fgl_lib
MAIN
    CALL fgl_lib.myfunction() -- function defined in fgl_lib
    CALL myfunction() -- function defined in this module
END MAIN
FUNCTION myfunction()
```



```
...
END FUNCTION
```

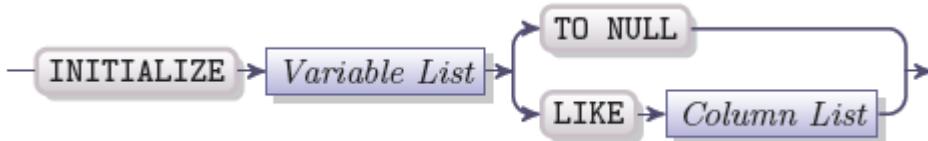
The same rules apply to referencing the variables or constants. Variables or constants declared with the use of the PUBLIC keyword can be referenced by their names. If several imported modules have variables or constants with the same names or the current module contains variables with the matching names, you can use the name of the imported module as the prefix to distinguish them.

```
IMPORT fgl_lib1
IMPORT fgl_lib2
DEFINE var INT
MAIN
    LET var = 100
    LET fgl_lib1.var = 200
    LET fgl_lib2.var = 300
...
END MAIN
```



INITIALIZE

The INITIALIZE statement is used to assign initial values to program variables. It can also assign a NULL value.



Element	Description
Variable List	A list of variables of simple data types, RECORD data type or array elements separated by commas
Column list	A list of database columns with table qualifiers, if necessary, separated by commas

Any variable declared with the help of the DEFINE statement has storage space allocated to it. The value assigned to a variable occupies that space. The INITIALIZE can be used with:

- The LIKE keyword, which assigns the default values of a database column to a variable (they are retrieved from the syscolval system table).
- The TO NULL keywords, which assign NULL value to a variable (the representation of a NULL value depends on the data type declared).

Variable List

The variable list can include one or more of the following variable types separated by commas:

- Variables of simple data types
- Variables of RECORD data types in the following formats:
 - *record.**
 - *record.first_member THRU record.last_member*
 - *record.member*
- Elements of static or dynamic program arrays in the format *array[dimensions]* where "dimensions" are from one to three integer expressions that specify the position of the array element.

The LIKE Clause

The LIKE keyword in the INITIALIZE statement is used to specify the default values from syscolval columns. It requires the default database to be specified, thus the DATABASE keyword must appear before the first program block of the program module where you use the INITIALIZE statement with the LIKE clause.

The variables specified in the variable list must match the database columns specified in the column list in order, number and data types. A column must be prefixed with the name of its table.

```
INITIALIZE first_v, second_v
    LIKE my_table.col1, my_table.col2
```



If you use the `table.*` notation, you will assign the default values of every column of a table to the specified variables. The example below assigns the values of all the columns of a table to all the members of a record:

```
INITIALIZE my_record.* LIKE my_table.*
```

In an ANSI-compliant database you must specify not only the table name but also its owner, for the tables that are not owned by the current user. In the example below, the third column belongs to the current user, so the `owner` table qualifier is omitted:

```
INITIALIZE first_v, second_v, third_v  
    LIKE j_smith.table1.col1, t_brown.table2.col5, table3.col7
```

You can include the owner table qualifier in a non ANSI-compliant database, but it is not obligatory. However, an incorrect owner prefix will produce an error.

The default values of columns are retrieved from the syscolval table, any changes in this table are reflected in a program (i.e. in the INITIALIZE statement), only if they are made before the program has been compiled. To apply the changes made after that, you must recompile the program. If a column has no default value, NULL value is assigned to an initialized variable.



Note: You cannot use the LIKE keyword to assign values to the variables of large data types (BYTE and TEXT). They can be assigned only NULL values.

The TO NULL Clause

The TO NULL clause is used to assign NULL values to variables. The examples below initialize the variables and all record members to NULL:

```
INITIALIZE first_v, second_v, third_v TO NULL  
INITIALIZE my_rec.* TO NULL
```

The variables are usually initialized to NULL due to such reasons:

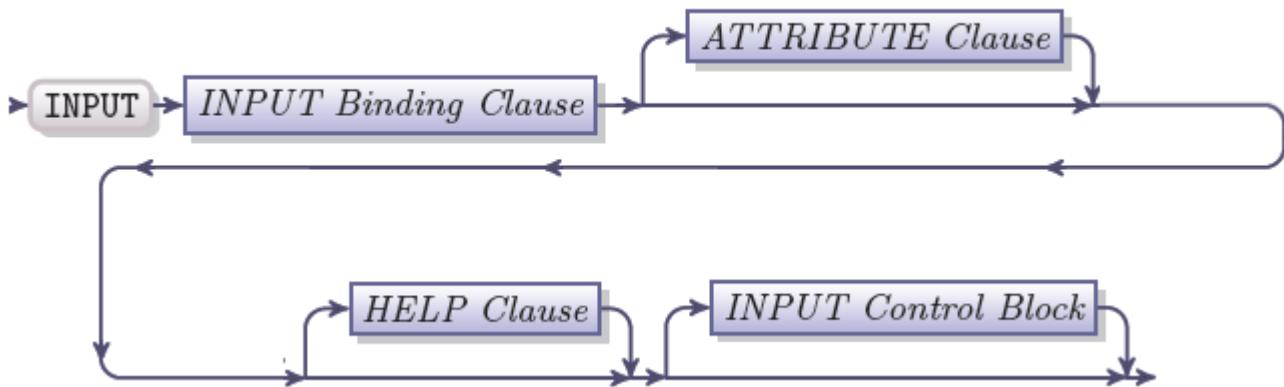
- To initialize variables that do not have the assigned value
- To discard the value assigned to a variable (it may prove useful if you plan to reuse this variable)

In order to optimize performance, you may want to limit the use of the INITIALIZE statement with the TO NULL clause.



INPUT

The INPUT statement is used to perform data entry into form fields.



Element	Description
INPUT Binding Clause	The clause that associates the variables with the input fields, It is the only obligatory clause
ATTRIBUTE Clause	An optional ATTRIBUTE clause where the attributes for the fields used for input can be specified
HELP Clause	An optional HELP clause where the help number for the input can be specified
INPUT Control Block	An optional block which controls the process of input

The INPUT statement assigns the values entered into form fields to the corresponding program variables. This statement may include clauses that specify the actions which 4GL should perform before or after input, clauses that are executed only under certain conditions (such as cursor movement or when some actions are performed by the user), etc.

To use the INPUT statement, follow these steps:

1. Create a form file and specify one or more text fields in this file, compile the form file.
2. Declare program variables by means of the DEFINE statement
3. Open the form either with the help of the OPEN FORM and DISPLAY FORM statements or with the help of the OPEN WINDOW ... WITH FORM statement.
4. Use the INPUT statement to assign values entered into the form fields to the variables.

When the INPUT statement is encountered, it triggers the following actions:

1. The default values are displayed to the form fields. The values are not displayed if:
 - o there are no default values assigned to these fields (you may assign default value to a field using the form specification file)
 - o you use the INPUT statement with the WITHOUT DEFAULTS
2. The cursor is moved to the first field (the order of fields is specified by the INPUT statement either explicitly or implicitly)
3. The user is allowed to perform input either in any order or in the order specified in the INPUT control clause



4. The value entered by the user is assigned to the corresponding variable when the cursor leaves the field, or if the Accept key is pressed.

When 4GL encounters the INPUT statement, the most recently displayed form or the form in the current window is activated. It is deactivated when the input is complete.

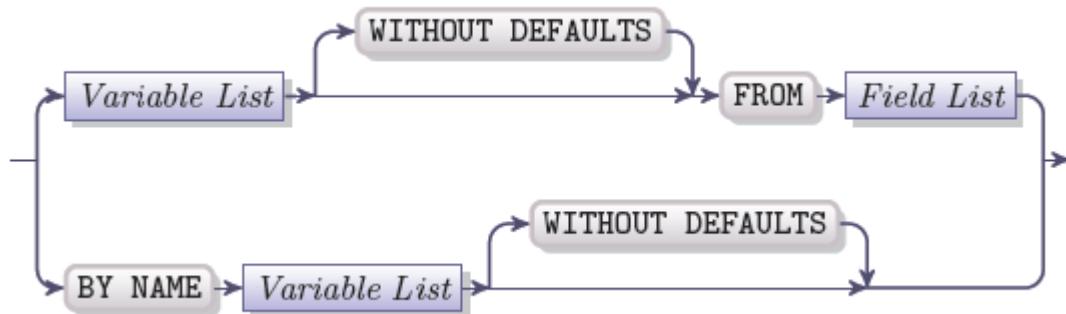
The INPUT statement must always have the Binding clause, where the form fields are associated with the program variables; it can also have such optional clauses:

- ATTRIBUTE clause
- HELP clause
- INPUT control clauses:
 - BEFORE INPUT
 - BEFORE FIELD
 - ON KEY, ON ACTION
 - AFTER FIELD
 - AFTER INPUT

The INPUT statement can include the CONTINUE INPUT and EXIT INPUT statements. If the INPUT statement contains at least one INPUT control clause, the end of the INPUT statement must be indicated with the help of the END INPUT keywords.

The INPUT Binding Clause

An INPUT statement must contain a Binding clause which correlates the program variables with the form fields. It temporary binds them to enable the data manipulation.



Element	Description
Variable list	The list of variables that are associated with the form fields
Field List	The list of form fields used for input

The variable list must include one or more variables separated by commas. The variables in the variable list support the syntax of the receiving variable of the LET statement. You can use variables of simple data types, records or their members with the THRU/THROUGH keyword and .* notation or individual array elements.

The field list of the FROM clause can include any fields that are declared in the form specification file (formonly fields, fields bind to table columns, members of the screen records or elements of screen arrays).



	Note: The elements of a program array cannot be used in the variable list. To perform input into them use the INPUT ARRAY statement.
---	---

There are two types of the Binding clause:

1. The general case of the Binding clause of the following pattern: "INPUT *variable-list* FROM *field-list*".
2. The special case where the names of variables in the variable list and the names of the fields in the field list are the same. It is the "INPUT BY NAME *variable-list*" type.

Variables and Fields Correlation

The total number of the program variables used in the INPUT statement must be the same as the total number of form fields that are specified explicitly by the FROM clause or implicitly by the BY NAME clause.

The order in which the cursor is moved from one field to another depends on the order of the fields in the FROM clause or on the order of the fields implied by the BY NAME clause. It is also influenced by the WRAP and FIELD ORDER options of the closest preceding OPTIONS statement.

The data types of fields and corresponding variables must be the same or compatible. 4GL checks whether the data type of the entered data is correct against the data type of the variable, it does not check the correspondence of the data type against the form field.

The variables used in the INPUT statement can be of any data type. However, fields that are associated with the database columns of SERIAL data type are skipped while performing input.

Default Values

The default values for the form fields will be displayed in them, if the INPUT statement contains no WITHOUT DEFAULTS keywords. The default values can be assigned to a form field in the form specification file:

- In the text form editor: by adding the DEFAULT attribute to the field declaration in the ATTRIBUTES section

```
ATTRIBUTES
...
f002 = formonly.f002,
DEFAULT = "default value"
```

- In the graphical form editor: by editing the default value of a selected field in "Field" tab of the Properties View.

The NULL value is assigned to all variables that do not have the default values set. However, if some of the fields associated with the columns of the DATE, DATETIME, INTERVAL and MONEY data types do not have the default values, the NULL value is not assigned to them. Below is the list of the default values assigned to some data types by 4GL, if the default value is not specified by the programmer:

Data Type of a Field	Default value assigned by 4GL
----------------------	-------------------------------



	automatically
Character data types	White space (=ASCII 32)
Numeric data types	0
INTERVAL	0
MONEY	\$0.00
DATE	12/31/1899
DATETIME	1899-12-31 23:59:59.99999

The WITHOUT DEFAULTS Keywords

If the WITHOUT DEFAULTS keywords are present in the INPUT statement:

- the default field values are not displayed
- the current values of the variables of the *variable-list* of the INPUT statement are displayed to the corresponding fields

The WITHOUT DEFAULTS keywords can appear in both types of the Binding clause (with and without the BY NAME keywords).

If the INPUT statement contains the WITHOUT DEFAULTS keywords, the user sees the current values of the variables or values of the table columns in the current row and it is possible to change these values by entering other values to the form fields. The built-in function field_touched() can help to validate which fields have been changed and thus to define which columns in a table require to be updated.

The BY NAME Clause

The BY NAME clause is used when the variable names and the field names are the same; it implicitly associates variables with the fields that have corresponding names. When 4GL compares the names of the variables with the names of the fields all the prefixes (i.e. record names) are ignored. The INPUT BY NAME statement works correctly, if the names of the variable (and of the corresponding form fields) are unique within the scope of their reference. A runtime error will occur if they are not unique.

Only those fields are active for data entry, which are specified implicitly by the BY NAME clause. If you use the BY NAME clause, you cannot include the FROM clause into your INPUT statement.

The FROM Clause

The FROM clause is used in an INPUT statement without the BY NAME clause, when the field names are not the same as the variables names. The cursor can be placed only in the fields specified in the FROM clause, the next and previous fields are predefined by the order of the fields in the FROM clause. The fields and the variables must match in order and number, they must be of the same or compatible data types.

```
INPUT my_progresc.address, my_progresc_phonenumb FROM f004, f005
```

The THRU/THROUGH keywords may be used to indicate a set of the ordered record members between the two specified members (inclusive):

```
INPUT my_progresc.lname THRU my_progresc.addit_inform FROM f001, f002, f003,
f004, f005, f006, f007
```

You can also use the .* notation to simplify the FROM clause if you create a screen record that includes all the fields which are used for input and the input is performed into all the members of a program record:



```
INPUT my_progres.* FROM my_screenrec.*
```



Note: The THRU/THROUGH keyword cannot be used in the FROM clause.

The ATTRIBUTE Clause

The Attribute clause of the INPUT statement has the general structure and features described in the [ATTRIBUTE clause](#) section of this reference guide.

The attributes specified in the ATTRIBUTE clause of the INPUT statement are applied only when the form is active. When form is inactive, it reverts to the previous attributes.

```
INPUT BY NAME fname, lname, address, phone  
      ATTRIBUTE (BLUE, REVERSE)
```

The ATTRIBUTE clause of the INPUT statement overrides attributes specified in the ATTRIBUTE clauses of the DISPLAY FORM, OPTIONS, or OPEN WINDOW statements for the same form.

The HELP Clause

The HELP clause consists of the HELP keyword and the literal integer which specifies the number of the help message associated with the INPUT statement.



Element	Description
Integer Expression	A 4GL expression that returns a positive integer greater than 0 and less than 32,767.

The help message appears in the help window, if the help key is pressed during the input. By default the help key is CONTROL-W, the default settings can be changed by means of the OPTIONS statement.

If the user presses the help key in the example below, it will result in displaying the help message 78, provided that the cursor is in any field that belongs to the screen record *scr_rec*.

```
INPUT pr_rec.* FROM scr_rec.* HELP 78
```

The help messages and their numbers are specified in a help file. You tell 4GL what help file to use by including the HELP FILE "*help-file name*" in the OPTIONS statement. The *help-file name* must be specified with the extension. You must specify the compiled help file, pay attention that the help file will have different extensions depending on whether it is compiled or not. (For more information see the "OPTIONS" statement section).

A runtime error will occur if:

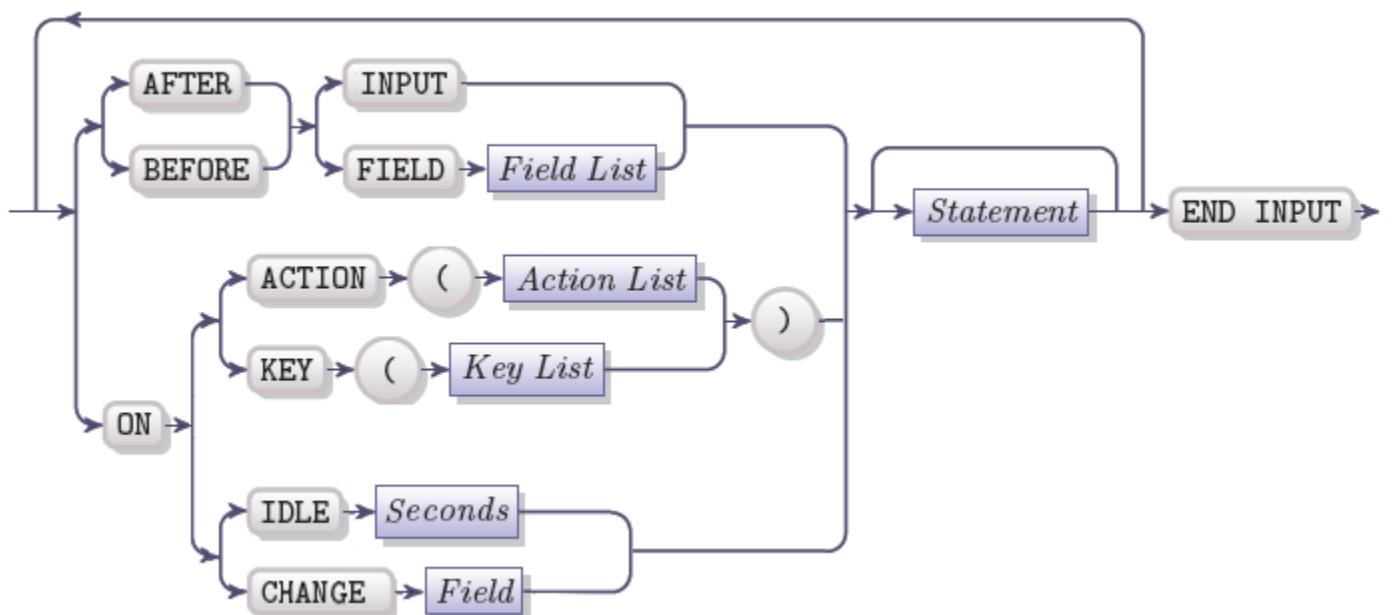


- the specified help file cannot be opened
- the help number specified in the HELP clause does not exist in the help file
- the help number is less than -32,767 and greater than 32,767.

The help message applies to the entire INPUT statement, however, you can specify a separate help message for a specific field, to do so specify the help key in an ON KEY block that contains the infield() operator and showhelp() function. If you want to display different help messages for each field, the messages should be displayed not in a separate help window but in the 4GL screen or window.

The INPUT Control Clauses

The INPUT control clauses specify the behaviour of the cursor and keys during the input, etc. Each INPUT control clause must contain at least one executable statement and the keywords that specify when this statement or statements will be executed.



Element	Description
Field List	A list of one or more form fields separated by commas
Action Clause	The ON ACTION clause with all its possible variants
Key Clause	The ON KEY clause with all its possible variants
Statement	The statement block of the INPUT control clause

An INPUT control clause can specify:

- Actions to be performed before and after the input (BEFORE INPUT and AFTER INPUT clauses)
- Actions to be performed before and after the specified field (BEFORE FIELD and AFTER FIELD clauses)
- Actions to be performed when user presses a key or a button (ON KEY and ON ACTION clauses)

The BEFORE FIELD and AFTER FIELD clauses as well as the NEXT FIELD keywords refer to the order of fields specified explicitly by the FROM clause or implicitly by the BY NAME clause.



If the INPUT statement contains at least one INPUT control clause, the END INPUT keywords must be included into the INPUT statement. If there are no INPUT control clauses, 4GL waits until the data are entered to the fields specified in the INPUT statement. It terminates if the Accept key is pressed.

The Statement Block

The statement block following each INPUT control clause must include at least one executable statement. It can include the following:

- 4GL or SQL statements
- The NEXT FIELD keywords that specify the next field where the cursor will be moved
- The CONTINUE INPUT keywords that return control to the user without INPUT termination
- The EXIT INPUT keywords that terminate the INPUT statement

While the statements in a control clause are executed, the form becomes inactive. It is reactivated after all the statements within the clause have been executed.

The Order of the INPUT Control Clauses

The INPUT control clauses can be listed in any order. However, 4GL executes them in the following order:

1. BEFORE INPUT
2. BEFORE FIELD
3. ON KEY, ON ACTION
4. AFTER FIELD
5. AFTER INPUT

You can have any number of the ON KEY and ON ACTION clauses, the maximum number of both the BEFORE FIELD and AFTER FIELD clauses is limited by the number of fields specified by the INPUT statement. You can have only one BEFORE INPUT and one AFTER INPUT clause within the INPUT statement.

Within the control clauses you can use the NEXT FIELD, EXIT INPUT and CONTINUE INPUT keywords, which are described further in this chapter.

The BEFORE INPUT Clause

The BEFORE INPUT block contains statements that are executed before the form is activated for the data input. It may be used to display messages that contain instructions on how to perform the input.

```
INPUT pr_rec.* FROM scr_rec.*  
    BEFORE INPUT  
        DISPLAY "To complete input press ECS" AT 2, 2
```

This clause is executed after the default values have been displayed, but before the user is allowed to enter something to the fields. If fields have no default values, NULL values will be displayed. If the Binding clause of the INPUT statement contains the WITHOUT DEFAULTS keywords, the current values of the variables associated with the fields will be displayed. If no values have been assigned to the variables yet, NULL values will be displayed. There can be only one BEFORE INPUT clause in an INPUT statement. You cannot include the infield() operator in this control clause.



The BEFORE FIELD Clause

The statements of the BEFORE FIELD clause are executed when the cursor moves to the specified field but before the user is allowed to enter any data into this field. Each field can have no more than one BEFORE FIELD clause.

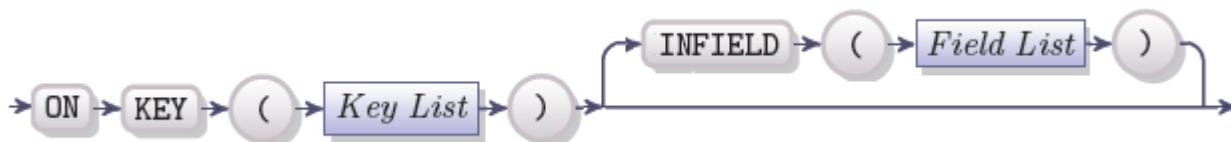
If you want to restrict the access to a field, use the NEXT FIELD keywords in the BEFORE FIELD clause. You can also display a default value to the field with the help of the DISPLAY statement in this control clause.

```

INPUT var1, var2, var3 FROM f001, f002, f003
    BEFORE FIELD f001
        IF var1 > 100 THEN
            DISPLAY "The maximal value for ", var1, "is reached."
        END IF
        NEXT FIELD f002
        BEFORE FIELD f002
        DISPLAY "1002" TO f002
    ...
END INPUT

```

The ON KEY Clauses



Element	Description
Field List	A list of one or more form fields separated by commas
Key List	A list that consists of one to four key names separated by commas

The ON KEY clauses can be included into the INPUT statement to specify the behaviour of the keys pressed while the input is performed. The keys can be assigned to form widgets and the actions will be triggered when such widget is pressed. However, if a key combination is not assigned to any widget, it can be activated with the help of the keyboard. One INPUT statement can contain any number of the ON KEY clauses, they can be placed in any order.

Infield Option

The ON KEY clause of the INPUT and INPUT ARRAY statements can include an optional `infield()` operator. This operator specifies a field or a list of fields for which the ON KEY clause will be triggered, if the key referenced by it is pressed. If the key referenced by the ON KEY clause is pressed, and the cursor is located in one of the fields specified in the `infield()` operator, the statements contained in such ON KEY clause will be executed. However, if the cursor is not located in any of the fields specified in the field list of the `infield()` operator, the ON KEY clause will be ignored, even if the corresponding key is pressed.

If the `infield()` operator is omitted, the ON KEY clause will be triggered when the referenced key is pressed regardless of the cursor position. If one and the same INPUT statement has two ON KEY clauses referencing the same key, but one of them has the optional `infield()` operator, the ON KEY... INFIELD() will be executed



when the cursor is in one of the referenced field and the specified key is pressed, in all other cases the ON KEY clause without the `infield()` operator will be executed.

In the example below the global ON KEY event will be triggered, if the cursor is positioned in field f1 or f4 at the moment when F5 is pressed. Otherwise, the field specific event will be triggered and the global event will be ignored.

```
INPUT BY NAME f1,f2,f3,f4
ON KEY(F5)
CALL fgl_messagw_box("Global ON KEY event")
ON KEY(F5) INFIELD(F2,F3)
CALL fgl_messagw_box("ON KEY event for fields f2 and f3")
END INPUT
```

Allowed Keys

The ON KEY clause can use one or several of the following keys separated by commas enclosed in parentheses to specify the key to which the other statements in the ON KEY clause refer:

ESC/ESCAPE	ACCEPT
NEXT/ NEXTPAGE	PREVIOUS/PREVPAGE
INSERT	DELETE
RETURN	TAB
INTERRUPT	HELP
LEFT	RIGHT
UP	DOWN
F1 – F256	CONTROL- <i>char</i>

Any character can be used as *char* in the combination CONTROL-*char* except the following characters: A, D, H, I, J, K, L, M, R, and X. The key names can be entered either in lower case or in upper case letters.

Any 4GL or SQL statements can be used in the ON KEY clause, it can also include EXIT INPUT, CONTINUE INPUT and NEXT FIELD keywords. These statements are executed when the key specified in the corresponding ON KEY clause is pressed.

When the user presses a key and the corresponding ON KEY clause within the INPUT statement is activated, the following actions are performed:

1. The input is suspended
2. The characters entered into the current field before the key has been pressed are stored in the input buffer
3. The statements that belong to the corresponding ON KEY clause are executed
4. The characters saved in the buffer are restored to the field
5. The input is resumed, the cursor is placed at the end of the character string retrieved from the buffer

The default behaviour of the ON KEY clause can be changed, if you add the NEXT FIELD in this clause, which will specify the field where the input should be resumed.

The contents of the input buffer can be changed by assigning a new value to the variable by means of the statements in an ON KEY clause. To support specific field actions use the `infield()` operator in this clause.



```

INPUT var1, var2, var3 FROM f001, f002, f003
    ON KEY (CONTROL-R, F14)
        CALL client_info()
    ON KEY (CONTROL-P)
        IF var1 IS NOT NULL AND var2 IS NOT NULL THEN
            CALL insert_data()
        END IF
    
```

Some keys require special attention if used in the ON KEY clause:

Key	Usage Features
ESC/ESCAPE	If you want to use this key in ON KEY clause, you must specify another key as the Accept key in the OPTIONS block, because ESCAPE is the Accept key by default
INTERRUPT	DEFER INTERRUPT statement must be executed in order that this key could be used in the ON KEY clause. On pressing the INTERRUPT key the corresponding ON KEY clause is executed, int_flag is set to non-zero but the CONSTRUCT statement is not terminated
QUIT	DEFER QUIT statement must be executed in order that this key could be used in the ON KEY clause. On pressing the QUIT key the corresponding ON KEY block is executed and int_flag is set to non-zero
CTRL-char (A, D, H, L, R, X)	4GL reserves these keys for field editing and they should not be used in the ON KEY clause
CTRL-char (I, J, M)	These key combinations by default mean TAB, NEWLINE and RETURN. If they are used in the ON KEY clause, they cannot perform their default functions while the OK KEY block is functional. Thus, if you use these keys in the ON KEY clause, the period of time during which this clause is functional should be restricted.

Some restrictions in key usage might be applied depending on your operational system. Many systems use such key combinations as CONTROL-C, CONTROL-Q, and CONTROL-S for the Interrupt, XON, and XOFF signals.

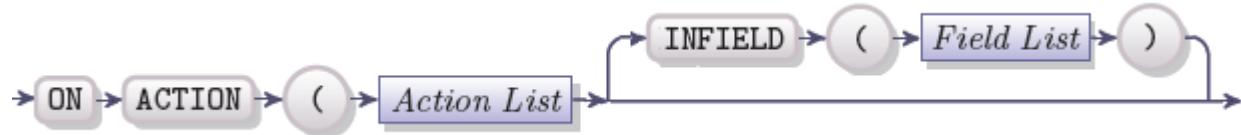
If the default Accept and Help keys are redefined with the help of the OPTIONS statement, the new Accept and Help keys cannot be used in the ON KEY clauses. If you define the F1 key to be the help key, you cannot use the F1 key in an ON KEY clause.

```
OPTIONS HELP KEY (F1)
```

After all the statements within the ON KEY block have been executed, the form is reactivated, the cursor is in the same field where it has been before the key specified in an ON KEY block has been pressed, unless the ON KEY clause contains the EXIT INPUT or the NEXT FIELD keywords.



The ON ACTION Clauses



Element	Description
Field List	A list of one or more form fields separated by commas
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks

The actions which you have assigned to the buttons in the form specification file can be used in an ON ACTION clause. The ON ACTION clauses are treated by 4GL very much like the ON KEY clauses. 4GL executes the statements specified in it, when the user presses a button or other widget on the form to which the corresponding action is assigned. Whereas an ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard, an ON ACTION clause is activated with the help of a form widget.

An event (action) can be assigned to the following widgets: button, radio button, check box, function field, and hotlink. It must be entered into the corresponding field in the Graphical Form Editor without quotation marks and white spaces. In the text form editor it must be enclosed into quotation marks:

```
CONFIG = "action {Name of the Button}"
```

The actions in an ON ACTION clause must be represented by a character string enclosed both in quotation marks and in parentheses and it must be the same as the name of the action in the form specification file:

```
ON ACTION ("action")
```

Lycia II also supports an in-built action named "*doubleclick*". The statements specified in the ON ACTION ("doubleclick") control block are invoked when the user performs a double click of the left mouse button at any place of the application window.

```
ON ACTION ("doubleclick")
```

It is also possible to set different doubleclick actions for different form widgets. To do this, one should add the INFIELD clause and specify the field or fields, double clicking on which should invoke the statements in the ON ACTION block:

```
ON ACTION ("doubleclick") INFIELD (f001)
```

There can be any number of the ON ACTION clauses in an INPUT statement. Up to four actions enclosed in quotation marks and separated by commas may be included into one ON ACTION clause.

As well as with the ON KEY clause, any 4GL or SQL statements can be included into an ON ACTION clause. The statements of a corresponding ON ACTION clause are executed when the widget within a screen form to which the action is assigned is pressed. The form where the INPUT statement takes place is deactivated while the statements are executed, though the statement is not terminated and the form will be reactivated, when 4GL finishes executing the ON ACTION clause, unless it contains the EXIT INPUT statement. If the ON ACTION clause contains the EXIT INPUT statement, the INPUT statement will be terminated.



The example below represents an ON ACTION clause which is triggered when the button with the "default" event is pressed:

```
ON ACTION ("default")
CALL display_default_values()
```

Infield Option

The ON ACTION clause of the INPUT and INPUT ARRAY statements can include an optional infield() operator. This operator specifies a field or a list of fields for which the ON ACTION clause will be triggered. If the action referenced by the ON ACTION clause is triggered, and the cursor is located in one of the fields specified in the infield() operator, the statements contained in such ON ACTION clause will be executed. However, if the cursor is not located in any of the fields specified in the field list of the infield() operator, the ON ACTION clause will be ignored, even if the corresponding action is triggered.

If the infield() operator is omitted, the ON ACTION clause will be executed when the referenced action is triggered regardless of the cursor position. If one and the same INPUT statement has two ON ACTION clauses referencing the same action, but one of them has the optional infield() operator, the ON ACTION... INFIELD() will be executed when the cursor is in one of the referenced fields and the specified action is triggered, in all other cases the ON ACTION clause without the infield() operator will be executed.

In the example below the ON ACTION event will be triggered, if the cursor is positioned in field f4 and the action is triggered. If the cursor is located in any field except f4, the ON ACTION clause will be ignored and nothing will happen.

```
INPUT BY NAME f1,f2,f3,f4
ON ACTION("test_action") INFIELD(f4)
CALL fgl_message_box("ON KEY event for field f4")
END INPUT
```

The ON IDLE Clauses



Element	Description
Seconds	An integer value specifying the time period in seconds

The ON IDLE clause is used to specify the actions that will be taken if the user performs no actions during the specified period of time.

The *seconds* parameter should be represented by an integer value or a variable which contains such a value. If the value is specified as zero, the input timeout is disabled.

It is advised that you specify the *seconds* parameter as a relatively long period of time (more than 30 seconds), because shorter delays may be caused by some external situations that distract the user from the application, so, the control block will be executed inappropriately:

```
INPUT chck FROM f002
```



```
ON IDLE 120
CALL fgl_message_box("You've been out for 2 minutes")
END INPUT
```

The ON CHANGE Clauses



Element	Description
Field	An integer value specifying the time period in seconds

The program executes the ON CHANGE program block after the cursor goes to another field and the value of the field specified in the block has changed **and** the field is marked as touched.

If the ON CHANGE actions are specified for a radio box, combo box, spin edit or slider form widget, the control block execution starts just after the user changes their value. With other widgets, the ON CHANGE block is executed when the cursor leaves them.

If a field has an ON CHANGE and AFTER FIELD blocks specified for it, the ON CHANGE block is executed first. If an ON ACTION or ON KEY block has changed the value of a field with an ON CHANGE block, and this value differs from the reference value and the field is marked as a touched one, the ON CHANGE block is executed after the cursor goes to the next field. It is advised that one doesn't perform field validation actions within an ON CHANGE block.

The AFTER FIELD Clause

Each field can have only one AFTER FIELD clause which will be executed every time the cursor leaves the corresponding field. To leave the field any of the following keys can be used:

- Home key
- End key
- RETURN/ENTER or TAB key
- Accept key
- Interrupt key
- Quit key

The two last keys can be used in this situation only if the DEFER INTERRUPT or the DEFER QUIT statement is in effect (respectively).

```
INPUT var1, var2, var3, var4 FROM f001, f002, f003, f004
...
AFTER FIELD f002
    IF var1>var2 THEN
        NEXT FIELD f004
    END IF
END INPUT
```



By default, the INPUT statement is terminated when the Accept key is pressed, while cursor is in any field, or when the RETURN or TAB key is pressed, when it is in the last field. To override these settings, include the NEXT FIELD keywords into the AFTER FIELD clause.



Note: If each field of the form has an AFTER FIELD clause which contains the NEXT FILED keywords, the user is unable to leave the INPUT statement.

The AFTER INPUT Clause

The AFTER INPUT clause is executed when the INPUT statement is terminated. One INPUT statement can have only one AFTER INPUT clause. It is typically used to save, validate or change the entered data.

The AFTER INPUT clause is executed when the INPUT statement is terminated by means of pressing:

- The Accept key
- The Interrupt key (with the DEFER INTERRUPT statement in effect)
- The Quit key (with the DEFER QUIT statement in effect)

The AFTER INPUT clause is **not** executed when the INPUT statement is terminated by means of pressing:

- The Interrupt key (if the DEFER INTERRUPT statement is **not** in effect)
- The Quit key (if the DEFER QUIT statement is **not** in effect)

It will **not** be executed also, if the EXIT INPUT statement has been executed in one of the optional clauses

```
INPUT BY NAME var1, var2, var3, var4
...
AFTER INPUT
    IF int_flag THEN
        IF var1 IS NULL THEN
            NEXT FIELD var1
        END IF
    END IF
END INPUT
```

The NEXT FIELD keywords in this clause return the cursor to the specified field. These keywords may be used in the AFTER INPUT clause only in a conditional statement (i.e the IF statement), otherwise the user will be unable to exit the INPUT statement.

The NEXT FIELD Keywords

The NEXT FIELD keywords specify the field to which the cursor will be moved. If no NEXT FIELD keywords are specified, the cursor will be moved to the next field according to the field order set by the Binding clause.



The user moves form field to field with the help of the TAB, RETURN and arrow keys. The NEXT FIELD keywords change the default movement of the screen cursor.

The NEXT FIELD keywords can be followed by:

- NEXT keyword, which will move the cursor to the next field:

```
NEXT FIELD NEXT
```

- PREVIOUS keyword, which will move the cursor to the previous field:

```
NEXT FIELD PREVIOUS
```

- The name of the field to which the cursor will be moved:

```
NEXT FIELD field-name
```

In the following example the field *var1* should be filled so if it contains NULL value when the cursor is moved to another field, the NEXT FIELD clause returns the cursor to this field:

```
INPUT BY NAME var1, var2, var3, var4
...
AFTER FIELD var1
    IF var1 IS NULL THEN
        DISPLAY "You should enter data in this field" AT 2,2
        NEXT FIELD var1
    END IF
END INPUT
```

4GL does not execute the statements within an optional INPUT control clause that follow the NEXT FIELD keywords and moves the cursor immediately to the specified field. In the example below, the *function_1()* is not called if the IF condition is met:

```
INPUT BY NAME var1, var2, var3, var4
...
AFTER FIELD var1
    IF var1 IS NULL THEN
        DISPLAY "You should enter data in this field" AT 2,2
        NEXT FIELD var1
    END IF
    CALL function_1 ()
END INPUT
```



The NEXT FIELD keywords may appear in any INPUT control clause, they usually appear within conditional statements as in the example above, but they may be used outside the conditional statements. The NEXT FIELD keywords can be used in a conditional statement within the BEFORE FIELD clause, if you want to restrict the access to this field.

	<p>Note: Be aware that if the NEXT FIELD keywords occur outside a conditional statement within the AFTER INPUT clause, the user will be unable to leave the INPUT statement.</p>
--	---

The CONTINUE INPUT Keywords

When 4GL encounters the CONTINUE INPUT statement, it skips all the statements that follow these keywords within the INPUT control clause. The control is returned to the recently occupied field. This statement is useful when you want to return control to the user from the execution of the deeply nested conditional statements. The CONTINUE INPUT statement may also prove useful in the AFTER INPUT control block that is used to examine the contents of the input buffer, the cursor can be returned to the form with the help of the CONTINUE INPUT statement, if the value needs to be changed.

The EXIT INPUT Keywords

The EXIT INPUT statement can be located in any of the INPUT control clauses. When 4GL encounters this statement, it performs the following actions:

- All the statements between the EXIT INPUT and END INPUT statements are skipped
- The form is deactivated
- The statements that follow the END INPUT keywords are executed

The AFTER INPUT clause is ignored, if the EXIT INPUT statement is executed.

The END INPUT Keywords

The END INPUT keywords mark the end of the INPUT statement. They are required if the INPUT statement contains at least one INPUT control clause and should be placed after the last INPUT control clause.

Cursor Movement Control

The user can position the visual cursor during the input with the help of the keyboard. Some keys are sensitive to the type of a field (a simple field, a segment of a multi-segment field).

Simple Fields

The cursor is moved from one simple field to another in the order specified explicitly by the FROM clause or implicitly by the BY NAME clause.

To change the position of the screen cursor the user can press arrow keys:

Arrow key	Action
↓	DOWN ARROW moves the cursor to the next field. With the FIELD ORDER UNCONSTRAINED option in the OPTIONS statement, the cursor is moved to the field below the current field. If there is no field below the current field, the cursor is moved to the field on the right, if any.



↑	UP ARROW moves the cursor to the previous field. With the FIELD ORDER UNCONSTRAINED option in the OPTIONS statement, the cursor is moved to the field above the current field. If there is no field below the current field, the cursor is moved to the field on the left, if any.
←	LEFT ARROW moves the cursor one space to the left within the field without erasing or changing the contents of the field. If the cursor is at the beginning of the field, it is moved to the previous field. The effect is the same as of the key CONTROL-H.
→	RIGHT ARROW moves the cursor one space to the right within the field without erasing or changing the contents of the field. If the cursor is at the end of the field, it is moved to the beginning of the next field. The effect is the same as of the key CONTROL-L.

The values entered in the fields that do not have the NOENTRY attribute can be edited while the INPUT statement is in effect by pressing the following keys:

Key	Action
CONTROL-A	It is used to select either insert mode (inserting new characters between the existing ones at the place of the cursor location) or type-over mode (replacing the existing characters with those which are entered)
CONTROL-D	Deletes characters to the right of the cursor position and till the end of the field
CONTROL-H	Moves the cursor one character to the left
CONTROL-L	Moves the one character to the right
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes character beneath the cursor

Multi-Segment Fields

The multi-segment fields resemble screen arrays, though the successive lines are just the segments of the same field and not screen records.

If the data is too long to be displayed in one line of this field (a line of a multi-segment field is called a segment), 4GL tries to separate the character string at the place of a white space character, if possible, and continue displaying the rest of the string in the second segment of the multi-segment field. This process is repeated until all the characters are displayed, if the field has enough segments.

When input or editing is performed in such field, the characters will be moved down to the segments below if necessary, due to the WORDWRAP field attribute. Blank characters used at the end of each segment are called editor blanks. These blanks can be prevented from being stored in the database with the help of the COMPRESS keyword in the form specification file.

When the cursor is in a multi-segment field, additional options for the cursor management become available. The CONTROL-J should be pressed to start NEWLINE. The RETURN key moves cursor to the next field, not to the next line of the multi-segment field.

When values are entered to a multi-segment field, the keys described below can be used to move the cursor over the data. The cursor will not stop at the editor blanks. It is not possible to overwrite the NEWLINE symbol. If a NEWLINE character is encountered by a cursor in the type-over mode, the type-over mode will be automatically changed to the insert mode and the new characters will be inserted between the last symbol of the current line and the NEWLINE symbol.



When the cursor enters a multi-segment field, the editing mode is set to the type-over mode. It can be moved with the help of the following keys:

Arrow key	Action
RETURN	The cursor is moved to the next field (not to the next line of the current multi-segment field)
TAB	The Tab character is entered and the cursor is moved to the next tab stop. The following text may jump right to align at a tab stop. It moves the cursor to the next tab stop that falls on an intentional character, if the type-over mode is active. The cursor can be moved to the next line, if necessary
↓	DOWN ARROW moves the cursor to the next field, if it has been located at the last segment of a multi-segment field. If it has been located at any segment but the last one, the cursor is moved to the segment below the current segment and is positioned at the same column. The cursor will be shifted to the left to avoid a tab symbol or an editor blank, if any
↑	UP ARROW moves the cursor to the previous field, if it has been located at the first segment of the multi-segment field. If it has been located at any segment but the first one, the cursor is moved to the segment above the current segment and is positioned at the same column. The cursor will be shifted to the left to avoid a tab symbol or an editor blank, if any
← or BACKSPACE	LEFT ARROW or BACKSPACE moves the cursor one character to the left. If the cursor is at the left end of a segment, it moves to the right end of the previous segment. If the cursor is at the left end of the first segment, the cursor is moved to the first character of the previous field (if the INPUT WRAP in the OPTIONS statement is in effect), or it will beep (if INPUT NO WRAP in the OPTIONS statement is in effect)
→	RIGHT ARROW moves the cursor one character to the right. If the cursor is located at the last character of the segment, it is moved to the first character of the next segment. If the cursor is located at the last character of the last segment, it is moved to the first character of the next field or beeps (this depends on the INPUT WRAP/NO WRAP option activated in the OPTIONS statement)

When any data is entered to a segment of a multi-segment field, all the characters located to the right of the cursor are shifted to the right and can be moved down to the segments below. If the total length of the character string exceeds the total length of all the segments, the excessive characters at the right end of the last segment are discarded.

The data entered to a multi-segment field can be edited with the help of the following keys:

Key	Action
CONTROL-A	Used to select either insert mode (inserting new characters between the existing ones at the place of the cursor location) or type-over mode (replacing the existing characters with those which are entered)
CONTROL-D	Deletes characters to the right of the cursor position and till the end of the multi-segment field (till the end of its last segment)
CONTROL-J	Inserts a NEWLINE character. The subsequent text aligns at the first column of the next segment. The excessive characters at the right end of the last segment, if any, will be discarded.
CONTROL-X	Deletes character beneath the cursor, this may pull up the characters from the segments below
CONTROL-H	Moves the cursor one character to the left, it can also move the cursor to



	the last character of the previous segment from the first character of the current segment
CONTROL-L	Moves the cursor one character to the right, it can also move the cursor to the first character of the next segment from the last character of the current segment
CONTROL-R	Redisplays the screen

Large Data Types Used in the INPUT Statement

The large data types are displayed by the 4GL in the following way:

- TEXT values – 4GL can display as much of the TEXT value a form field can contain
- BYTE values – 4GL cannot display these values in a form field, the <BYTE value> character string will be displayed instead.

The WORDWRAP editor cannot display the TEXT values; multi-segment fields can display as much of the text value as they can accommodate.

If the PROGRAM attribute is specified in the form specification file for the fields of BYTE or TEXT data types, the user can view the values of these fields by means of pressing the exclamation mark key (!), when the cursor is positioned in such field. The values will be viewed through an external editor, which allows the user to edit text or graphics.

When the external editor is opened, all the keys specified in the ON KEY or ON ACTION clauses are ignored. After the external editor is terminated, 4GL restores the 4GL screen to the state in which it has been before opening the editor, resumes INPUT at the same field, and reactivates the ON KEY and ON ACTION clauses.

Completing the INPUT Statement

The INPUT statement can be terminated by means of pressing:

- Accept key
- Interrupt key
- Quit key
- RETURN or TAB key when the cursor is at the last field of the form

It will also be terminated, if 4GL executes the EXIT INPUT statement.

Pressing either of the above listed keys deactivates the form. The Quit and Interrupt keys also terminate the program unless the DEFER INPUT or DEFER INTERRUPT statement is in effect.

The user must press the Accept key to finish the input if:

- The INPUT WRAP option is specified in the OPTIONS statement
- The AFTER FIELD clause for the last field contains the NEXT FIELD keywords

If a DEFER INTERRUPT statement has been executed previously, pressing Interrupt key will result in the following actions:

- The global variable **int_flag** will be set to TRUE
- The INPUT statement will be terminated but the program will be still running



If a DEFER QUIT statement has been executed previously, pressing Quit key will result in the following actions:

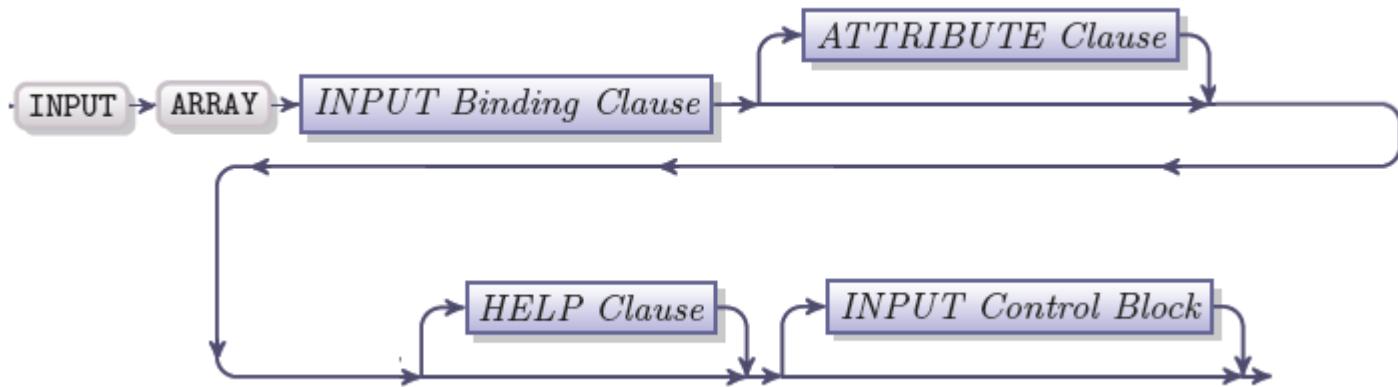
- The global variable **quit_flag** will be set to TRUE
- The INPUT statement will be terminated but the program will be still running

When the INPUT statement is terminated, 4GL executes first the AFTER FIELD clause for the current field, if any, and then the AFTER INPUT clause, if it is present. These clauses are not executed when the user terminates the INPUT statement by pressing the Interrupt or Quit key, or if 4GL encounters the EXIT INPUT statement.



INPUT ARRAY

The INPUT array is a special case of the INPUT statement which allows user to input data from a screen array into a program array.



Element	Description
INPUT Binding Clause	The clause that associates the variables with the input fields, It is the only obligatory clause
ATTRIBUTE Clause	An optional ATTRIBUTE clause where the attributes for the fields used for input can be specified
HELP Clause	An optional HELP clause where the help number for the input can be specified
INPUT Control Block	An optional block which controls the process of input

The INPUT ARRAY statement is used to assign values entered to a screen array to one or more record of a program array. Optional clauses can be included into this statement to control the process of input. To use the INPUT ARRAY statement:

1. Create a form file and specify a screen array in it, compile the form file.
2. Declare an ARRAY OF RECORD by means of the DEFINE statement
3. Open the form either with the help of the OPEN FORM and DISPLAY FORM statements or with the help of the OPEN WINDOW ... WITH FORM statement.
4. Use the INPUT ARRAY statement to assign values entered into the screen array to the program array elements.

When the INPUT statement is encountered, it triggers the following actions:

1. The default values are displayed to the form fields. The values are not displayed if:
 - o there are no default values assigned to these fields (you may assign default value to a field in the form specification file)
 - o you use the INPUT statement with the WITHOUT DEFAULTS keywords
2. The cursor is moved to the first field (the order of fields is specified by the Binding clause of the INPUT statement)



3. The value entered by the user is assigned to the corresponding variable when the cursor leaves the field or if the Accept key is pressed.

When 4GL encounters the INPUT ARRAY statement, the most recently displayed form or the form in the current window is activated. It is deactivated when the input is complete.

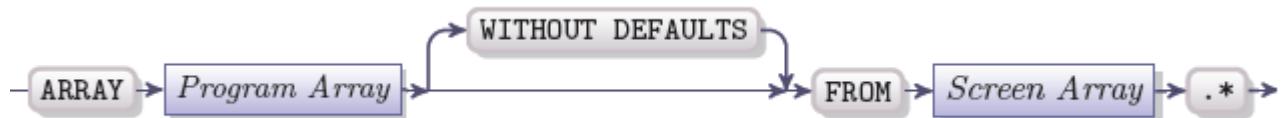
The INPUT ARRAY statement must always have the Binding clause, where the screen array fields are associated with the program array elements; it can also have such optional clauses:

- ATTRIBUTE clause
- HELP clause
- INPUT ARRAY control block

The INPUT ARRAY statement can include the CONTINUE INPUT and EXIT INPUT statements. If the INPUT ARRAY statement contains at least one INPUT control clause, the end of the INPUT ARRAY statement must be marked with the help of the END INPUT keywords.

The INPUT ARRAY Binding Clause

An INPUT ARRAY statement must contain the Binding clause which correlates the program array with the screen array. It temporary binds them to enable the data manipulation.



Element	Description
Program Array	The name of a variable of ARRAY or DYNAMIC ARRAY data type which will store values supplied during the input
Screen Array	The name of the screen array where the input will be performed

The Correlation of Variables and Fields

The FROM clause, which is required and cannot be omitted, associates the screen array with the program array. There can be other fields in the form which are not included into the screen record.

The fields in each row of a screen array and members in each record of the program array should match in number, order and be of the same or compatible data types. The entered data are validated against the data type of the record members of the program array but not against the data types of the fields.

The variables used as record members of a program array can be of any data type. However, a variable declared LIKE a database column of SERIAL data type is skipped while performing input.

The number of the records in the program array determines how many program records the array can store. The number of records within the screen array specifies the number of rows that can be displayed by the form at a time. If the size of the program array greater than the size of the screen array, use the Next page and Previous page keys to see the records that do not fit into the displayed screen record.



By default, the cursor moves over the screen array in the order defined by the program array declaration. At the beginning of the input the cursor is located at the first screen array of the screen record.

The Default Values

The default values for the form fields will be displayed, if the INPUT statement contains no WITHOUT DEFAULTS keywords. The default values are assigned to a form field in the form specification file:

- In the text form editor: by adding the DEFAULT attribute to the field declaration in the ATTRIBUTES section

```
ATTRIBUTES
...
f002 = formonly.f002,
DEFAULT = "1234"
```

- In the graphical form editor: by editing the default value in "Field" tab of the form properties.

The NULL value is assigned to all variables that do not have a default value set. However, if some of the fields are associated with the columns of the DATE, DATETIME, INTERVAL and MONEY data type which do not have the default values, the NULL value is not assigned to them as the default value. Here are the default values assigned to some data types by 4GL, if the default value is not specified by the programmer:

Data Type of a Field	Default value assigned by 4GL automatically
Character data types	White space (=ASCII 32)
Numeric data types	0
INTERVAL	0
MONEY	\$0.00
DATE	12/31/1899
DATETIME	1899-12-31 23:59:59.99999

The WITHOUT DEFAULTS Keywords

If the WITHOUT DEFAULTS keywords are present in the INPUT ARRAY statement:

- the default field values are not displayed
- the current values of the member variables of the record of a program array are displayed to the screen array fields

To display the previously initialized values do the following:

1. INITIALIZE variables
2. Call the built in function set_count() to determine the number of rows that are stored in the program array
3. Use the INPUT ARRAY with the WITHOUT DEFAULTS keywords

When the INPUT ARRAY statement contains the WITHOUT DEFAULTS keywords, the user sees the current values of the variables or values of the table columns in the current row and it is possible to change these

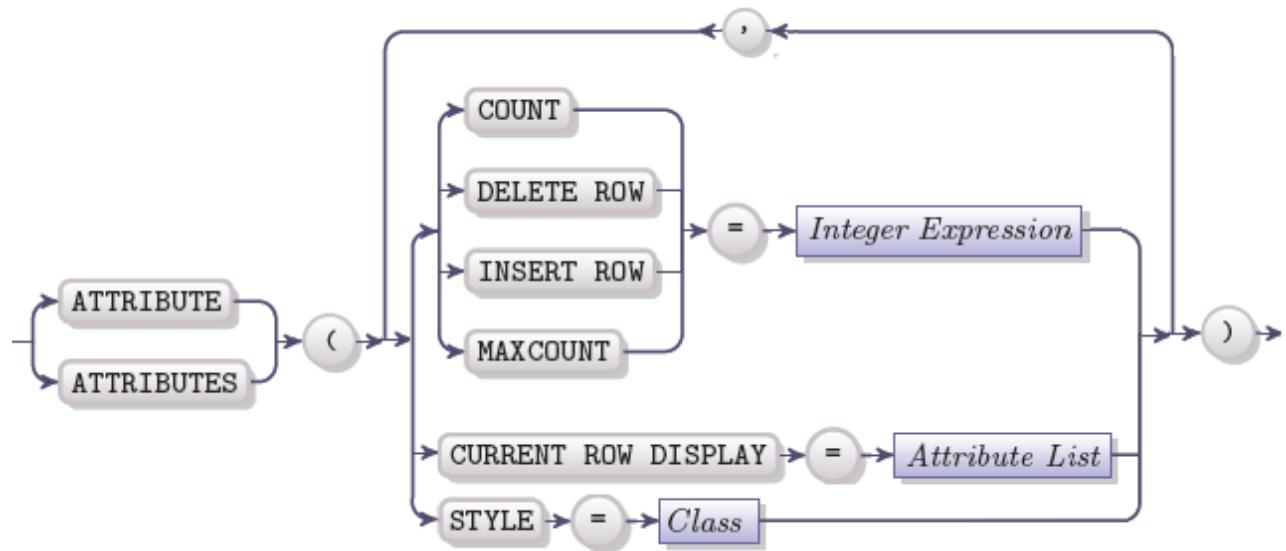


values by entering other values to the form fields. The built-in function field_touched() can help to validate which fields have been changed and thus to define which columns in a table require update.

The ATTRIBUTE Clause

The Attribute clause of the INPUT ARRAY statement has the general structure described in the [ATTRIBUTE clause](#) section of this reference. However, it can include the specific attributes. The CURRENT ROW DISPLAY attribute, which specifies the display attributes to the current row of the screen array. All the other attributes influence all the fields of the screen array.

Below is the syntax of the attributes unique for the INPUT ARRAY clause, these are not all the attributes available to the INPUT ARRAY statement, they can be complimented by the colour and intensity attributes:



Element	Description
Integer Expression	The integer expression that returns a positive integer
Attribute List	The list of general display attributes
Class	The character value specifying a name of a theme class

The attributes specified in the ATTRIBUTE clause of the INPUT ARRAY statement are applied to the form only when it is active.

```

INPUT ARRAY my_arr FROM my_scrarr.*  

    ATTRIBUTE (BLUE, REVERSE)
  
```

The ATTRIBUTE clause of the INPUT ARRAY statement overrides attributes specified in the ATTRIBUTE clauses of the DISPLAY FORM, OPTIONS, or OPEN WINDOW statements for the same form.

The CURRENT ROW DISPLAY Attribute

To highlight the current row of the screen array, you may use the special attribute CURRENT ROW DISPLAY which is valid in the ATTRIBUTE clause of the INPUT ARRAY statement, but not in the INPUT statement.

The attributes enclosed in the quotation marks apply only to the currently highlighted row of the screen array. The following example highlights the current row of the *sa_account* screen array:



```
INPUT ARRAY l_account_arr FROM sa_account.*  
ATTRIBUTE (CURRENT ROW DISPLAY = "YELLOW, REVERSE")
```

There should be at least one attribute within the quotation marks.

The COUNT Attribute

The COUNT attribute can be used to specify the quantity of the program records within the program array that contain data. This attribute can be used only in the ATTRIBUTE clause of the INPUT ARRAY statement.

```
ATTRIBUTE (COUNT = 10)
```

The effect of the COUNT attribute is equivalent to the effect of the built-in function set_count(). The COUNT attribute overrides the value of the set_count() function called previously.

The COUNT and MAXCOUNT attributes support integer expressions or variables that return integers unlike the other attributes that support only corresponding keywords or literal integers.

The MAXCOUNT Attribute

The MAXCOUNT attribute is used to define the dynamic size of a screen array. The size specified in this attribute can differ from the actual size of the screen array; it can be smaller than the size of the screen array defined in the form specification file. This attribute can be used only in the ATTRIBUTE clause of the INPUT ARRAY statement.

Both COUNT and MAXCOUNT attributes can be specified within one INPUT ARRAY statement.

```
INPUT ARRAY l_account_arr FROM sa_account.*  
ATTRIBUTE (COUNT=a, MAXCOUNT=b)
```

The "a" and "b" can be either literal integers or variables of INTEGER data type, "a" determines the maximal number of program records stored in the program array *l_account_arr* and "b" specifies the maximum size of the screen array *sa_account*.

If the MAXCOUNT value is greater than the actual size of the screen array, or if it is less than 1, the MAXCOUNT will be automatically specified the same size as the actual size of the screen array.

The COUNT and MAXCOUNT attributes support integer expressions or variables that return integers unlike the other attributes that support only corresponding keywords or literal integers.

The INSERT ROW Attribute

The Insert key can be enabled or disabled during the array input with the help of the INSERT ROW attribute. The INSERT ROW attribute can acquire the following values: 1 – for true or 0 – for false. You can also write TRUE or FALSE in words. The default value of this attribute is TRUE.

If the INSERT ROW attribute is set to FALSE, the Insert key is disabled and the user cannot perform insert actions while the INPUT ARRAY statement is in effect. However, the user can perform insert actions by means of the TAB, ARROW, and RETURN keys.



If the INSERT ROW attribute is set to TRUE, the user can use the Insert key to enter data.

```
INPUT ARRAY l_account_arr FROM sa_account.*  
ATTRIBUTE (INSERT ROW=FALSE)
```

The DELETE ROW Attribute

The Delete key can be enabled or disabled during the array input with the help of the DELETE ROW attribute. The DELETE ROW attribute can acquire the following values: 1 – for true or 0 – for false. You can also write TRUE or FALSE in words. The default value of this attribute is TRUE.

If the DELETE ROW attribute is set to FALSE, the Delete key is disabled and the user cannot perform delete actions while the INPUT ARRAY statement is in effect.

If the DELETE ROW attribute is set to TRUE, the user can use the Delete key to delete a row.

```
INPUT ARRAY l_account_arr FROM sa_account.*  
ATTRIBUTE (DELETE ROW=FALSE)
```

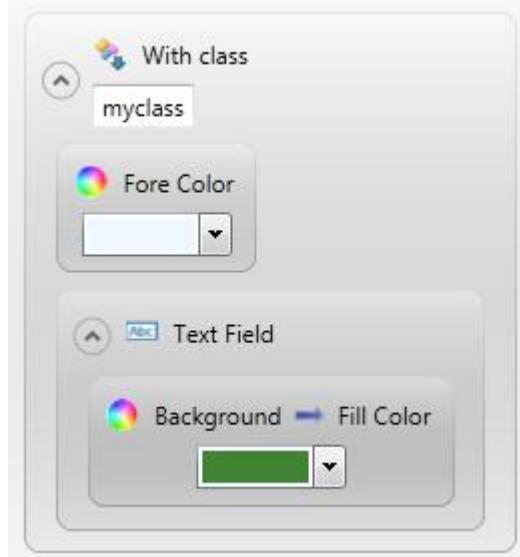
The STYLE Attribute

The STYLE attribute is used to bind the INPUT statement with a class specified in the current theme file. The theme file can include several classes, each having its own set of display properties. Therefore, the STYLE attribute can be used to set up context-depending input display settings.

The class name specified in the STYLE attribute, should be taken into quotes or passed as a variable value.
The following 4GL code:

```
INPUT ARRAY l_account_arr FROM sa_account.*  
ATTRIBUTE (STYLE = "myclass")
```

Together with the following *myclass* class settings:



Will give the result shown in the screenshot:



1-first	1-second	1-third
2-first	2-second	2-third
3-first	3-second	3-third
4-first	4-second	4-third

The HELP Clause

The HELP clause consists of the HELP keyword and the literal integer which specifies the number of the help message associated with the INPUT statement.



Element	Description
Integer Expression	A 4GL expression that returns a positive integer greater than 0 and less than 32,767.

The help message appears in the help window, if the help key is pressed during the input. By default the help key is CONTROL-W, the default settings can be changed by means of the OPTIONS statement.

If the user presses the help key in the example below, it will result in displaying the help message 78, provided that the cursor is in any field that belongs to the screen record *scr_ar*.

```
INPUT ARRAY pr_rec.* FROM scr_rec.* HELP 78
```

The help messages and their numbers are specified in a help file. You tell 4GL what help file to use by including the HELP FILE "*help-file name*" in the OPTIONS statement. The *help-file name* must be specified with the extension. You must specify the compiled help file, pay attention that the help file will have different extensions depending on whether it is compiled or not. (For more information see the "OPTIONS" statement section).

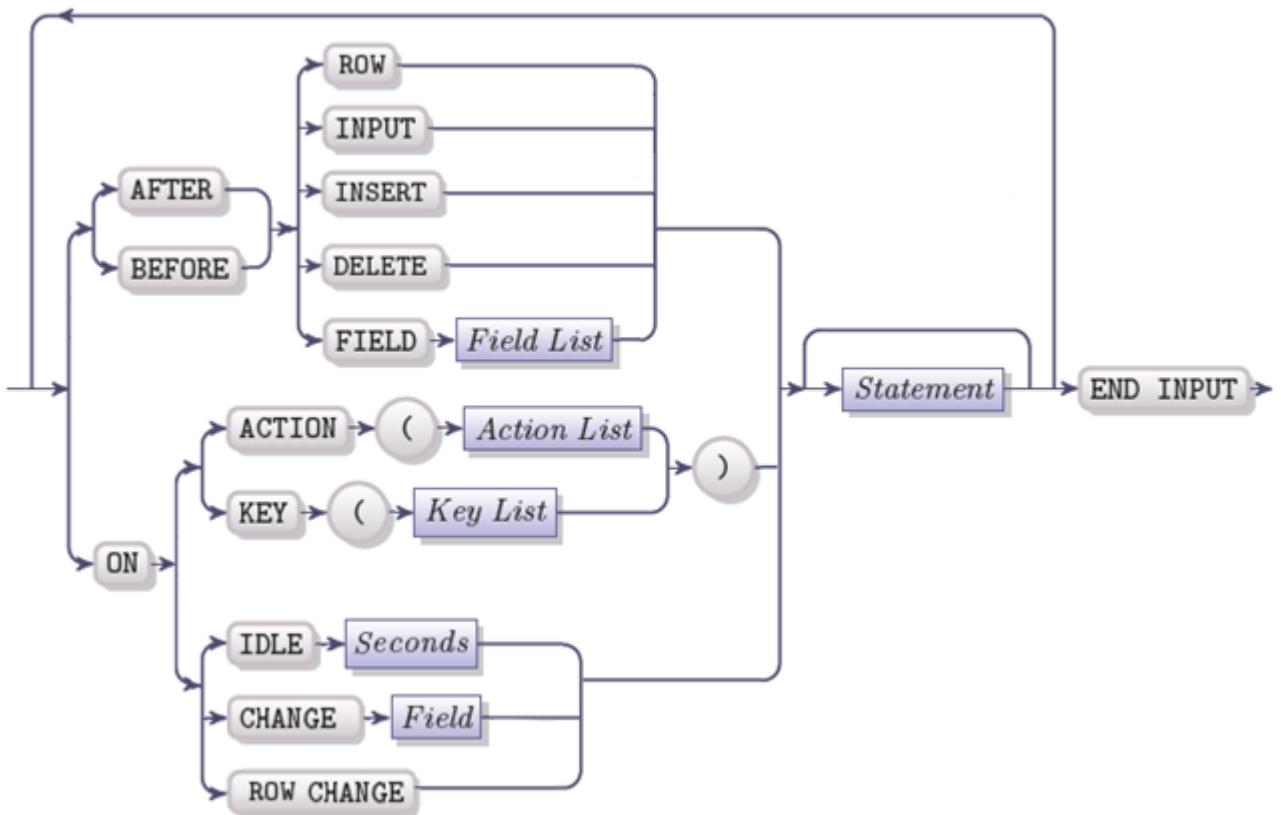
A runtime error will occur if:

- the specified help file cannot be opened
- the help number specified in the HELP clause does not exist in the help file
- the help number is less than -32,767 and greater than 32,767.

The help message applies to the entire INPUT statement, however, you can specify a separate help message for a specific field, to do so specify the help key in an ON KEY block that contains the *infield()* operator and *showhelp()* function. If you want to display different help messages for each field, the messages should be displayed not in a separate help window but in the 4GL screen or window.

The INPUT ARRAY Control Clauses

The INPUT ARRAY control clauses specify the behaviour of the cursor and keys during the input. Each INPUT ARRAY control clause must contain at least one executable statement and the keywords that specify when this statement or statements will be executed.



Element	Description
Field List	A list of one or more form fields separated by commas
Field	A single form field
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks
Key List	A list that consists of one to four key names separated by commas
Seconds	An integer value specifying the number of seconds
Statement	The statement block of the INPUT ARRAY control clause

An INPUT ARRAY control clause can specify:

- Actions to be performed before and after the input (BEFORE INPUT and AFTER INPUT clauses)
- Actions conditioned by the cursor movement (BEFORE FIELD and AFTER FIELD, BEFORE ROW and AFTER ROW clauses)
- Actions to be performed before and after the insert of a row (BEFORE INSERT and AFTER INSERT clauses)
- Actions to be performed before and after the delete of a row (BEFORE DELETE and AFTER DELETE clauses)
- Actions to be performed when user presses a key or a button (ON KEY and ON ACTION clauses)

If the INPUT statement contains at least one INPUT ARRAY control clause, the END INPUT keywords must be included into the INPUT ARRAY statement. If there are no INPUT ARRAY control clauses, 4GL waits until the data are entered to the fields specified in the INPUT ARRAY statement. It is terminated, if the Accept key is pressed.



The Statement Block

The statement block following each INPUT control clause must include at least one executable statement. It can include the following:

- 4GL or SQL statements
- The NEXT FIELD keywords that specify the next field where the cursor will be moved
- The CONTINUE INPUT keywords that returns control to the user without INPUT termination
- The EXIT INPUT keywords that terminate the INPUT statement

While the statements in a control clause are executed, the form becomes inactive. It is reactivated after all the statements within the clause have been executed.

The Field Order

The BEFORE FIELD and AFTER FIELD clauses as well as the NEXT FIELD keywords refer to the order of fields implied by the screen record in the FROM clause. You can specify the field with the help of a table reference or by prefixing it with the name of a record or an array : *array[/line]*.

The BEFORE FIELD *screen-array* and AFTER FIELD *screen-array* clauses influence the whole screen array, they are executed when the cursor enters or leaves any field that belongs to the screen array. The BEFORE FIELD *screen-array.field* and AFTER FIELD *screen-array.field* clauses reference a separate field within the screen array, they are executed when the cursor enters or leaves the specified field.

The Order of the INPUT ARRAY Control Clauses

The INPUT control clauses can be listed in any order. However, 4GL executes them in the following fixed order:

1. BEFORE INPUT
2. BEFORE ROW
3. BEFORE INSERT, BEFORE DELETE
4. BEFORE FIELD *screen-array*
5. BEFORE FIELD *screen-array.field*
6. ON KEY, ON ACTION, ON IDLE
7. ON CHANGE
8. AFTER FIELD *screen-array*
9. AFTER FIELD *screen-array.field*
10. AFTER INSERT, AFTER DELETE
11. ON ROW CHANGE
12. AFTER ROW
13. AFTER INPUT

You can have any number of the ON KEY and ON ACTION clauses, the maximal number of the BEFORE FIELD and AFTER FIELD clauses is limited by the number of fields in specified by the INPUT ARRAY statement. You can have only one BEFORE INSERT, AFTER INSERT, BEFORE DELETE, AFTER DELETE, BEFORE INPUT, AFTER INPUT, BEFORE ROW, and AFTER ROW clause.

Within the control clauses you can use the NEXT FIELD, EXIT INPUT and CONTINUE INPUT keywords, which are described further in this chapter.

The BEFORE INPUT Clause

The BEFORE INPUT block contains statements that are executed before the form is activated for the data input. It may be used to display messages that contain instructions on how to perform the input.



```
CALL set_count(1)
INPUT ARRAY pr_ar.* WITHOUT DEFAULTS FROM scr_ar.*
    BEFORE INPUT
        LET x = arr_curr()
        LET y = scr_line()
        LET pr_ar[x].lame = "Smith"
        DISPLAY pr_ar[x].lame TO scr_ar[y].lname
...
END INPUT
```

This clause is executed after the default values have been displayed to the fields, but before the user is allowed to enter something to the fields. If the fields have no default values, NULL values will be displayed. If the Binding clause of the INPUT ARRAY statement contains the WITHOUT DEFAULTS, the current values of the variables associated with the fields will be displayed. If the variables haven not been assigned values yet, NULL values will be displayed.

There can be only one BEFORE INPUT clause in an INPUT ARRAY statement. The field_touched() operator cannot be used in this clause.

The BEFORE ROW Clause

A screen record should be understood under ROW in the BEFORE ROW clause. One INPUT ARRAY statement can have only one BEFORE ROW clause.

It is executed when:

- the cursor is moved to the next row of the screen array
- the INSERT statement fails due to the lack of space
- the INSERT statement is terminated by the Interrupt key or the Quit key
- the Delete key is pressed

The BEFORE DELETE Clause

The BEFORE DELETE clause is executed after the user presses the Delete key during the INPUT ARRAY statement, and before the record is actually deleted. There can be only one BEFORE DELETE clause in an INPUT ARRAY statement.

If the EXIT INPUT is specified in the BEFORE DELETE clause, the row will not be deleted and the INPUT ARRAY statement will be terminated when the user presses the Delete key.

The statement clause of the BEFORE DELETE control block can include:

- executable 4GL statements
- SQL statements
- the EXIT INPUT keywords
- the CONTINUE INPUT keywords
- the CANCEL DELETE keywords



The CANCEL DELETE Keywords

The CANCEL DELETE keywords within the BEFORE DELETE clause can cancel either all the delete actions or the deleting can be cancelled for individual screen records of the currently displayed form file.

The cancelled delete operation does not affect the active set of rows that are being processed by the INPUT ARRAY statement.

The BEFORE DELETE clause is terminated when 4GL executes the CANCEL DELETE keywords and the program control is passed to the next clause following the BEFORE DELETE clause. You cannot specify the CANCEL DELETE keywords outside the BEFORE DELETE clause.

The CANCEL DELETE keywords of the BEFORE DELETE clause and the CANCEL INSERT keywords of the BEFORE INSERT clause have similar effect. As an example, the programmer might want the user to delete all but one of the rows, but once a row is deleted, a replacement row cannot be inserted in its place:

```
DEFINE n_rows INTEGER
DEFINE arrayname ARRAY[100] OF RECORD
...
INPUT ARRAY arrayname WITHOUT DEFAULTS FROM s_array.*
ATTRIBUTES (COUNT = n_rows, MAXCOUNT = n_rows,
INSERT ROW = FALSE, DELETE ROW = TRUE)
BEFORE INSERT
CANCEL INSERT
BEFORE DELETE
LET n_rows = n_rows - 1
IF n_rows <= 0 THEN
CANCEL DELETE
END IF
END INPUT
```

The BEFORE INSERT Clause

An INPUT ARRAY statement can have only one BEFORE INSERT clause. The BEFORE INSERT clause is executed when:

- the user begins inserting new records into the screen array
- the Insert key is pressed (it is used to insert one more record between the existing ones), but before the record is inserted
- The cursor is moved to a blank record at the end of a screen array

This clause is executed before the user is able to enter any data in the row he has inserted. The statement clause of the BEFORE INSERT control block can include:

- executable 4GL statements
- SQL statements
- the EXIT INPUT keywords
- the CONTINUE INPUT keywords
- the CANCEL INSERT keywords

The statement clause of the BEFORE INSERT control block can include special keywords CANCEL INSERT, which cancel the inserting of a new row and transfer the program control to the next control block.



The CANCEL INSERT Keywords

The CANCEL INSERT keywords within the BEFORE INSERT clause can cancel either all the insert actions or the inserting can be cancelled for individual screen records of the currently displayed form file. The cancelled insert operation does not affect the active set of rows that are being processed by the INPUT ARRAY statement.

The CANCEL INSERT keywords prevent used from inserting new rows by means of the Insert, TAB and RETURN key. The following example disables the Insert key for only the fifth row:

```
INPUT ARRAY ...
BEFORE INSERT
IF arr_curr() == 3
THEN
CANCEL INSERT
END IF
END INPUT
```

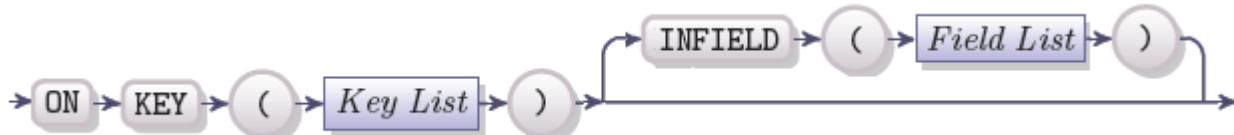
The BEFORE FIELD Clause

The statements of the BEFORE FIELD clause are executed when the cursor moves to the defined field but before the user is allowed to enter any data into this field. Each field can have no more than one BEFORE FIELD clause.

If you want to restrict the access to a field, use the NEXT FIELD keywords in the BEFORE FIELD clause. You can also display a default desired to the field with the help of the DISPLAY statement in this control clause.

```
INPUT ARRAY pr_ar.* WITHOUT DEFAULTS FROM scr_ar.*
BEFORE FIELD f001
DISPLAY "101" TO f001
...
END INPUT
```

The ON KEY Clauses



Element	Description
Field List	A list of one or more form fields separated by commas
Key List	A list that consists of one to four key names separated by commas

The ON KEY clauses can be included into the INPUT ARRAY statement to specify the behaviour of the keys pressed while the input is performed. The keys can be assigned to form widgets and the actions will be triggered when such widget is pressed. However, if a key combination is not assigned to any widget, it can



be activated with the help of the keyboard. One INPUT ARRAY statement can contain any number of the ON KEY clauses; they can be placed in any order.

Infield Option

The ON KEY clause of the INPUT and INPUT ARRAY statements can include an optional infield() operator. This operator specifies a field or a list of fields for which the ON KEY clause will be triggered, if the key referenced by it is pressed. If the key referenced by the ON KEY clause is pressed, and the cursor is located in one of the fields specified in the infield() operator, the statements contained in such ON KEY clause will be executed. However, if the cursor is not located in any of the fields specified in the field list of the infield() operator, the ON KEY clause will be ignored, even if the corresponding key is pressed.

If the infield() operator is omitted, the ON KEY clause will be triggered when the referenced key is pressed regardless of the cursor position. If one and the same INPUT ARRAY statement has two ON KEY clauses referencing the same key, but one of them has the optional infield() operator, the ON KEY... INFIELD() will be executed when the cursor is in one of the referenced field and the specified key is pressed, in all other cases the ON KEY clause without the infield() operator will be executed.

In the example below the global ON KEY event will be triggered, if the cursor is positioned in field f1 or f4 in any row of the screen array at the moment when F5 is pressed. Otherwise, the field specific event will be triggered and the global event will be ignored.

```
INPUT ARRAY my_arr FROM scr_arr.*  
ON KEY(F5)  
CALL fgl_messagw_box("Global ON KEY event")  
ON KEY(F5) INFIELD(F2,F3)  
CALL fgl_messagw_box("ON KEY event for fields f2 and f3")  
END INPUT
```

Allowed Keys

The ON KEY clause can use one or several of the following keys separated by commas enclosed in parentheses to specify the key to which the other statements in the ON KEY clause refer:

ESC/ESCAPE	ACCEPT
NEXT/ NEXTPAGE	PREVIOUS/PREVPAGE
INSERT	DELETE
RETURN	TAB
INTERRUPT	HELP
LEFT	RIGHT
UP	DOWN
F1 – F256	CONTROL- <i>char</i>

Any character can be used as *char* in the combination CONTROL-*char* except the following characters: A, D, H, I, J, K, L, M, R, and X. The key names can be entered either in lower case or in upper case letters.

Any 4GL or SQL statements can be used in the ON KEY clause, it can also include EXIT INPUT, CONTINUE INPUT and NEXT FIELD keywords. These statements are executed when the key specified in the corresponding ON KEY clause is pressed.



When an ON KEY clause within the INPUT ARRAY statement is activated, the following actions are performed:

1. The input is suspended
2. The characters entered into the current field before the key has been pressed are stored in the input buffer
3. The statements that belong to the corresponding ON KEY clause are executed
4. The characters saved in the buffer are restored to the field
5. The input is resumed, the cursor is placed at the end of the character string retrieved from the buffer

The default behaviour of the ON KEY clause can be changed, if you add the NEXT FIELD in this clause, which specifies the field where the input will be resumed.

The contents of the input buffer can be changed by means of assigning a new value to the variable using the statements of an ON KEY clause. To support specific field actions, use the `infield()` operator in this clause

```
INPUT ARRAY pr_ar.* WITHOUT DEFAULTS FROM scr_ar.*
    ON KEY (CONTROL-R, F14)
        CALL show_info()
    ON KEY (CONTROL-P)
        CALL us_help()
END INPUT
```

Some keys require special attention if used in the ON KEY clause:

Key	Usage Features
ESC/ESCAPE	If you want to use this key in ON KEY clause, you must specify another key as the Accept key in the OPTIONS statement, because ESCAPE is the Accept key by default
F1	It is the default Insert key, to use it in the ON KEY clause, specify another Insert key in the OPTIONS statement
F2	It is the default Delete key, to use it in the ON KEY clause, specify another Delete key in the OPTIONS statement
F3	It is the default Next key, if you want to use it in the ON KEY clause, you must specify another key as the Next key in the OPTIONS statement
F4	It is the default Previous key, if you want to use it in the ON KEY block, you must specify another key as Previous key in the OPTIONS block
INTERRUPT	The DEFER INTERRUPT statement must be executed so that this key could be used in the ON KEY clause. On pressing the INTERRUPT key the corresponding ON KEY clause is executed, <code>int_flag</code> is set to non-zero but the INPUT ARRAY statement is not terminated
QUIT	The DEFER QUIT statement must be executed so that this key could be used in the ON KEY clause. On pressing the QUIT key the corresponding ON KEY clause is executed and <code>int_flag</code> is set to non-zero value, but the INPUT ARRAY statement is not terminated
CTRL-char (A, D, H, L, R, X)	4GL reserves these key combinations for field editing and they should not be used in the ON KEY block
CTRL-char	These key combinations mean TAB, NEWLINE and RETURN. If they are used



(I, J, M)	in the ON KEY clause, they cannot perform their default functions while the ON KEY clause is functional. Thus, if you use these keys in the ON KEY clause, the period of time during which this block is functional should be restricted.
CTRL-char (W)	CONTROL-W is the default Help key, to use it in the ON KEY block, specify another Help key in the OPTIONS statement

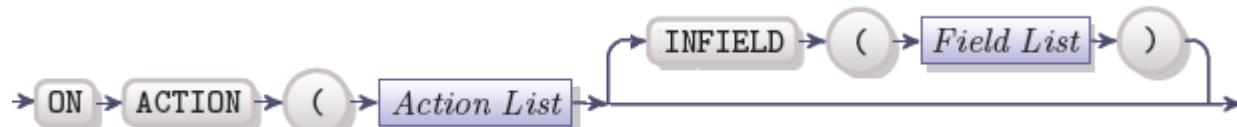
Some restrictions in key usage might be applied depending on your operational system. Many systems use such key combinations as CONTROL-C, CONTROL-Q, and CONTROL-S for the Interrupt, XON, and XOFF signals.

If the default Accept and Help keys are redefined with the help of the OPTIONS statement, the new Accept and Help keys cannot be used in the ON KEY clauses. If you define the F10 key as the Help key, you cannot use the F10 key in an ON KEY clause, but you can use the CONTROL-W key instead, which is not the Help key any more.

```
OPTIONS HELP KEY (F10)
```

After all the statements within the ON KEY clause have been executed, the form is reactivated, the cursor is in the same field where it has been before the key specified in an ON KEY block has been pressed, unless the ON KEY clause contains the EXIT INPUT or the NEXT FIELD keywords.

The ON ACTION Clauses



Element	Description
Field List	A list of one or more form fields separated by commas
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks

The actions which you have assigned to the buttons in the form specification file can be used in an ON ACTION clause. The ON ACTION clauses are treated by 4GL very much like the ON KEY clauses. 4GL executes the statements specified in it, when the user presses a button or other widget on the form to which the corresponding action is assigned. Whereas an ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard, an ON ACTION clause is activated with the help of a form widget.

An event (action) can be assigned to the following widgets: button, radio button, check box, function field, and hotlink. It must be entered into the corresponding field in the Graphical Form Editor without quotation marks and white spaces. In the text form editor, it must be enclosed into quotation marks:

```
CONFIG = "action {Name of the Button}"
```

The actions in an ON ACTION clause must be represented by a character string enclosed both in quotation marks and in parentheses and it must be the same as the name of the action in the form specification file:

```
ON ACTION ("action")
```



Lycia II also supports an in-built action named "*doubleclick*". The statements specified in the ON ACTION ("doubleclick") control block are invoked when the user performs a double click of the left mouse button at any place of the application window.

```
ON ACTION ("doubleclick")
```

It is also possible to set different doubleclick actions for different form widgets. To do this, one should add the INFIELD clause and specify the field or fields, double clicking on which should invoke the statements in the ON ACTION block:

```
ON ACTION ("doubleclick") INFIELD (f001)
```

There can be any number of the ON ACTION clauses in an INPUT ARRAY statement. Up to four actions enclosed in quotation marks and separated by commas may be included into one ON ACTION clause.

As well as with the ON KEY clause, any 4GL or SQL statements can be included into an ON ACTION clause. The statements of a corresponding ON ACTION clause are executed when the widget within a screen form to which the action is assigned is pressed. The form where the INPUT ARRAY statement takes place is deactivated during the statements are executed, though the statement is not terminated and the form will be reactivated, when 4GL finishes executing the ON ACTION clause, unless it contains the EXIT INPUT statement. If the ON ACTION clause contains the EXIT INPUT statement, the INPUT ARRAY statement will be terminated.

The example below represents an ON ACTION clause which is triggered when the button with the "default" event is pressed.

```
ON ACTION ("default")
CALL display_default_values()
```

Infield Option

The ON ACTION clause of the INPUT and INPUT ARRAY statements can include an optional `infield()` operator. This operator specifies a field or a list of fields for which the ON ACTION clause will be triggered. If the action referenced by the ON ACTION clause is triggered, and the cursor is located in one of the fields specified in the `infield()` operator, the statements contained in such ON ACTION clause will be executed. However, if the cursor is not located in any of the fields specified in the field list of the `infield()` operator, the ON ACTION clause will be ignored, even if the corresponding action is triggered.

If the `infield()` operator is omitted, the ON ACTION clause will be executed when the referenced action is triggered regardless of the cursor position. If one and the same INPUT ARRAY statement has two ON ACTION clauses referencing the same action, but one of them has the optional `infield()` operator, the ON ACTION... INFIELD() will be executed when the cursor is in one of the referenced fields and the specified action is triggered, in all other cases the ON ACTION clause without the `infield()` operator will be executed.

In the example below the ON ACTION event will be triggered, if the cursor is positioned in field f4 of any row of the screen array and the action is triggered. If the cursor is located in any field except f4, the ON ACTION clause will be ignored and nothing will happen.

```
INPUT ARRAY my_arr FROM scr_arr.*
ON ACTION("test_action") INFIELD(f4)
```



```
CALL fgl_messagw_box("ON KEY event for field f4")
END INPUT
```

The ON IDLE Clauses



Element	Description
Seconds	An integer value specifying the time period in seconds

The ON IDLE clause is used to specify the actions that will be taken if the user performs no actions during the specified period of time.

The *seconds* parameter should be represented by an integer value or a variable which contains such a value. If the value is specified as zero, the input timeout is disabled.

It is advised that you specify the *seconds* parameter as a relatively long period of time (more than 30 seconds), because shorter delays may be caused by some external situations that distract the user from the application, so, the control block will be executed inappropriately:

```
INPUT ARRAY my_arr FROM scr_arr.*
ON IDLE 120
CALL fgl_messagw_box("Yoy've been out for 2 minutes")
END INPUT
```

The ON CHANGE Clauses



Element	Description
Field	An integer value specifying the time period in seconds

The program executes the ON CHANGE program block after the cursor goes to another field and the value of the field specified in the block has changed **and** the field is marked as touched.

If the ON CHANGE actions are specified for a radio box, combo box, spin edit or slider form widget, the control block execution starts just after the user changes their value. With other widgets, the ON CHANGE block is executed when the cursor leaves them.

If a field has an ON CHANGE and AFTER FIELD blocks specified for it, the ON CHANGE block is executed first. If an ON ACTION or ON KEY block has changed the value of a field with an ON CHANGE block, and this value differs from the reference value and the field is marked as a touched one, the ON CHANGE block is executed after the cursor goes to the next field. It is advised that one doesn't perform field validation actions within an ON CHANGE block.

The AFTER FIELD Clause

Each field can have only one AFTER FIELD clause which will be executed every time the cursor leaves the corresponding field. To leave the field any of the following keys can be used:

- Arrow key



- RETURN or TAB key
- Accept key
- Interrupt key
- Quit key

The two last key can be used in this situation only if the DEFER INTERRUPT or the DEFER QUIT statement is in effect.

```
INPUT ARRAY pr_ar.* WITHOUT DEFAULTS FROM scr_ar.*  
...  
AFTER FIELD f002  
    IF var1>var2 THEN  
        NEXT FIELD f004  
    END IF  
END INPUT
```

By default the INPUT statement is terminated when the Accept key is pressed while the cursor is in any field or by pressing RETURN or TAB key when it is in the last field. To override these settings, include the NEXT FIELD keywords in the AFTER FIELD clause.



Note: If each field of the form has an AFTER FIELD clause which contains the NEXT FILED keywords, the user is unable to leave the INPUT statement.

The AFTER INSERT Clause

The AFTER INSERT clause is executed when the user inserts a screen record into a screen array. An INPUT ARRAY statement can have only one AFTER INSERT clause. A record can be inserted in the following way:

1. Enter the information in all the required fields of the inserted screen record
2. Move the cursor out of the record by pressing any arrow key, RETURN key, TAB key, Accept key, HOME key or END key.



Note: The insert is not considered to be finished and the AFTER INSERT clause is not executed until the user moves the cursor out of the last input field.

```
AFTER INSERT OF lname  
CALL show_list()
```

The AFTER DELETE Clause

The AFTER DELETE clause is executed when the user deletes values from a screen record by means of the Delete key. An INPUT ARRAY statement can have only one AFTER DELETE clause. If this clause is present, the following actions are performed when the Delete key is pressed:



1. The record is deleted from the screen array
2. The statements within the AFTER DELETE clause are executed
3. The statements within the AFTER ROW clause are executed, if the clause is present

The Accept key must also be pressed to make the corresponding modifications in the program array.

```
AFTER DELETE OF lname  
CALL show_list()
```

The ON ROW CHANGE Clause

The application executes the ON ROW CHANGE clause when the cursor leaves the current row and goes to another one. The ON ROW CHANGE clause is executed only if the user changed one or several values within the row and the touched flag of the corresponding fields are set. Note, that when the application leaves the dialog instruction or the input goes to another row, the touched flags for the current row are reset.

The ON ROW CHANGE clause can be used for dynamic modification of data in database, for changed data verification, etc.

If arr_curr() function is invoked from the ON ROW CHANGE clause, it returns the index of the current row in which the changes were performed.

```
INPUT ARRAY myarr WITHOUT DEFAULTS FROM scrarr.*  
ON ROW CHANGE  
LET i = arr_curr()  
MESSAGE "The row "||" was changed"  
END INPUT
```

The clause is executed prior to AFTER ROW clause.

The AFTER ROW Clause

The row within an INPUT ARRAY statement is a screen record, it is not necessarily associated with a database table. The AFTER ROW clause is executed when:

- The cursor is moved from the current row by means of pressing any arrow key, RETURN key, TAB key, Accept or Interrupt key (in the DEFER INTERRUPT statement is in effect).
- The Insert key has been pressed and a new screen record has been inserted.

Each INPUT ARRAY statement can have only one AFTER ROW clause. If both AFTER INSERT and AFTER ROW clauses are present within one INPUT ARRAY statement, the AFTER ROW clause is executed immediately after the AFTER INSERT clause.

An AFTER ROW clause can contain the NEXT FIELD keywords. When 4GL encounters these keywords, it moves the cursor to the next field in the next row, not in the row where the cursor has been before that. If the values of a row are in conflict, the cursor can be returned to the corresponding fields before completing the input.

```
INPUT ARRAY pr_ar.* WITHOUT DEFAULTS FROM scr_ar.*  
...  
AFTER ROW  
LET i=arr_curr()  
IF pr_ar[i].item_id <100 THEN
```



```
MESSAGE "You do not have access to products with ID less than 100"  
NEXT FIELD item_id  
END IF  
...  
END INPUT
```

The user cannot leave the *item_id* field until he enters the correct data to this field.

The AFTER INPUT Clause

The AFTER INPUT clause is executed when the INPUT ARRAY statement is terminated. One INPUT ARRAY statement can have only one AFTER INPUT clause. It is usually used to save, validate or change the entered data.

The AFTER INPUT clause is executed when the INPUT ARRAY statement is terminated. It is terminated when the user presses:

- The Accept key
- The Interrupt key (with the DEFER INTERRUPT statement in effect)
- The Quit key (with the DEFER QUIT statement in effect)

The AFTER INPUT clause is **not** executed when the INPUT ARRAY statement is terminated by pressing:

- The Interrupt key (if DEFER INTERRUPT statement is **not** effect)
- The Quit key (if DEFER QUIT statement is **not** effect)
- The EXIT INPUT statement has been executed in one of the optional clauses

```
INPUT ARRAY pr_ar.* WITHOUT DEFAULTS FROM scr_ar.*  
...  
AFTER INPUT  
    IF int_flag THEN  
        IF var1 IS NULL THEN  
            NEXT FIELD var1  
        END IF  
    END IF  
END INPUT
```

The NEXT FIELD keywords in the example above return the cursor to the specified field. These keywords may be used in the AFTER INPUT clause only in a conditional statement (i.e IF statement), otherwise the user will be unable to exit the INPUT statement.

The built-in functionality

The INPUT ARRAY supports some built-in functionality which allows to work with the data displayed to the screen array or to the tree view.

This functionality is bound to the screen grid and does not need any additional coding.



The Built-In Sorting

If the dialog is performed in a table, the user can sort the rows. Unlike with the DISPLAY ARRAY, row sorting in the INPUT ARRAY dialog cannot be applied to the tree views.

To sort the rows of a list, the user should click on the header of the column by which the sorting should be performed. A click on the header invokes a GUI event that makes the runtime system change the order of the rows displayed to the table.

Note, that the rows are sorted only visually and the data in the program array remain untouched. Therefore, after the rows were sorted this way, the current index of a row and its visual position may be different.

The system uses the standard collation to order the rows, and its behaviour depends on the locale settings. Therefore, the resulting order can vary from the order generated on DATABASE server with the ORDER BY clause of the SELECT statement.

When the user or the program closes the application window, the information about the sort column and order is passed to the user settings database of the system. This information will be retrieved and applied to the table when the window is opened again. Therefore, when the program restarts, the rows will be sorted according to the last settings. The saved settings are specific for each list container.

By default, the built-in sorting is enabled, but you can switch it off for a table or a tree by adding the UNSORTABLECOLUMNS attribute to the list container, or the UNSORTABLE attribute for the column/field. The UNSORTABLECOLUMNS attribute may be useful in the lists engaged in the INPUT ARRAY dialog.

The Built-In Find

The INPUT ARRAY statement supports the default built-in find feature, which includes implicitly invoked *find* and *findnext* actions. These actions can be treated as the regular ones.

The find action is triggered with the CONTROL-F keypress by default. The search can be performed in any list container (tables, trees, or grids). When the action is triggered, the program opens a popup dialog window to which the user should enter the search value. When the user enters the value and presses the OK button, the dialog starts searching a row in which a field value would match the entered one. After the first value is found, the user can press CONTROL-G (by default) to call the *findnext* action and to find the next matching in the list. The *findnext* action does not open the search dialog window, but resumes the search using the previously entered value.

The search value is compared with the string displayed to the fields, i.e. is based on the value format. For example, MONEY fields will have the currency symbol and the user should enter the search value including this symbol and using correct separators.

By default, the search is performed in all the columns, but the find dialog box offers the user to choose a specific column for search.

The search scans only the text widgets. Columns, using radio buttons, Checkboxes, Images, as well as TEXT and BYTE fields are not processed during the search. In dynamic tree views, the built-in find searches among the nodes stored in the program array.

You can specify your own actions for the *find* and *findnext* actions using the ON ACTION clause. In this case, the default built-in find action is disabled.

The NEXT FIELD Keywords

The NEXT FIELD keywords specify the field to which the cursor will be moved. If no NEXT FIELD keywords are specified, the cursor will be moved, by default, to the next field according to the field order set by the Binding clause. The user moves form field to field with the help of the TAB, RETURN and arrow keys. The NEXT FIELD keywords change the default movement of the screen cursor.

The NEXT FIELD keywords can be followed by:

- The NEXT keyword, which will move the cursor to the next field:



NEXT FIELD NEXT

- The PREVIOUS keyword, which will move the cursor to the previous field:

NEXT FIELD PREVIOUS

- The name of the field to which the cursor will be moved:

NEXT FIELD field-name

In the following example the field *var1* should be filled. So if it contains NULL value when the cursor is moved to another field, the NEXT FIELD clause returns the cursor to this field:

```
INPUT ARRAY arr1 FROM scr_arr1.*  
...  
ON ACTION ("verify")  
    IF infield(f002) THEN  
        CALL verif_f()  
        NEXT FIELD f003  
    END IF  
END INPUT
```

4GL does not execute the statements within an optional INPUT ARRAY control clause that follow the NEXT FIELD keywords and moves the cursor immediately to the specified field. In the example below the *function_1()* is not called if the IF condition is met:

```
INPUT ARRAY arr1 FROM scr_arr1.*  
...  
AFTER FIELD f001  
    IF var1 IS NULL THEN  
        DISPLAY "You should enter data in this field" AT 2,2  
        NEXT FIELD f001  
    END IF  
    CALL function_1 ()  
END INPUT
```

The NEXT FIELD keywords may appear in any INPUT ARRAY control clause, they usually appear within conditional statements as in the example above, but they may be used outside the conditional statements. The NEXT FIELD keywords can be used in a conditional statement within the BEFORE FIELD clause, if you want to restrict the access to this field.



	<p>Note: Be aware that if the NEXT FIELD keywords occur outside a conditional statement within the AFTER INPUT clause, the user will be unable to leave the INPUT ARRAY statement.</p>
---	---

To move the cursor from the last field of the form to the first field of the form use the NEXT FIELD keywords in the AFTER FIELD clause of the last field of the form. This can also be done by specifying the INPUT WRAP option in the OPTIONS statement.

The ACCEPT INPUT Keywords

The ACCEPT INPUT keywords make the application accept the inputted values and, if there are no errors, exits the INPUT ARRAY statement.

When the application encounters the ACCEPT INPUT keywords, it executes the ON CHANGE, AFTER FIELD blocks, etc., but skips all the statements following the ACCEPT INPUT keywords.

The values validation, triggered by ACCEPT INPUT keywords, is performed in several stages:

1. The program checks if the field values correspond to the program variable data types.
2. All input fields are checked against the NOT NULL field attribute, which demands the field to have some program or user specified data entered to it. If no value is entered at the moment of the input validation, the condition is considered not to be satisfied. Note, that if the value entered by the user consists only of spaces, the value is treated as NULL during the NOT NULL check.
3. The input fields are checked against the INCLUDE field attribute, which specifies the values, possible within the field. If the value specified in the field does not match any of the INCLUDE attribute values, the condition is considered to be unsatisfied.
4. All the input fields are checked against the REQUIRED field attribute, which makes it necessary for the field to have the default value of to have the touched flag triggered, either by user actions or program.

If a validated field does not satisfy at least one of the given requirements, the program does not exit from the INPUT ARRAY statement, an error message appears, and the program control returns to the first field that was not validated.

The CONTINUE INPUT Keywords

When 4GL encounters the CONTINUE INPUT statement it skips all the statements follow these keywords within the INPUT ARRAY control clause. The cursor is returned to the recently occupied field. This statement is useful, when you want to return control to the user from the execution of the deeply nested conditional statements. The CONTINUE INPUT statement may also prove useful in the AFTER INPUT control block that is used to examine the contents of the input buffer, the cursor can be returned to the form with the help of the CONTINUE INPUT statement, if the data need to be changed.

The EXIT INPUT Keywords

The EXIT INPUT statement can be located in any of the INPUT ARRAY control clauses. When 4GL encounters this statement, it performs the following actions:

- All the statements between the EXIT INPUT and END INPUT statements are skipped
- The form is deactivated



- The statements that follow the END INPUT keywords are executed

The AFTER INPUT clause is ignored, if the EXIT INPUT statement is executed.

The END INPUT Keywords

The END INPUT keywords mark the end of the INPUT ARRAY statement. They are required, if the INPUT ARRAY statement contains at least one INPUT ARRAY control clause. It should be placed after the last INPUT ARRAY control clause.

Cursor Movement and Data Edition

The user can position the visual cursor during the INPUT ARRAY statement with the help of the keyboard.

Cursor Movement

To change the position the screen cursor, the user can press the arrow keys:

Arrow key	Action
↓	The DOWN ARROW key moves the cursor one line down the screen to the same field in a row below the current row. If there is no row below the current one in the screen array, but there are still rows left in the program array below it, 4GL moves the information up one row. If the cursor is at the last row of the program array and the screen array, the message will appear which informs the user that there are no more rows in this direction.
↑	The UP ARROW key moves the cursor to the same field located one row above the current row. If the cursor has been on the first row of the screen array, but there are still rows left in the program array above it, 4GL moves the information down one row. If the cursor is at the first row of the program array and the screen array, the message will appear which informs the user that there are no more rows in this direction.
←	The LEFT ARROW key moves the cursor one space to the left within the field without erasing or changing the contents of the field. If the cursor is at the beginning of the field, it is moved to the previous field. The effect is the same as of the CONTROL-H key.
→	The RIGHT ARROW key moves the cursor one space to the right within the field without erasing or changing the contents of the field. If the cursor is at the end of the field, it is moved to the beginning of the next field. The effect is the same as of the CONTROL-L key.
F3	This key scrolls to the next full page of the program array records; this is the default Next key. This can be changed in the OPTIONS statement.
F4	This key scrolls to the previous full page of the program array records; this is the default Previous key. This setting can be changed in the OPTIONS statement.

The Editing Keys

The values entered in the fields that do not have the NOENTRY attribute can be edited while the INPUT ARRAY statement is in effect by pressing the following keys:

Key	Action
CONTROL-A	It is used to select either insert mode (inserting new characters between



	the existing ones at the place of the cursor location) or type-over mode (replacing the existing characters with those which are entered)
CONTROL-D	Deletes characters to the right of the cursor position and till the end of the field
CONTROL-H	Moves the cursor one character to the left
CONTROL-L	Moves the cursor one character to the right
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes the character beneath the cursor

Clearing Reserved Lines

The Error line is cleared, when the cursor is moved to a new field. The Error line is reserved for displaying messages specified in the ERROR statement, it is the last line of the 4GL window and 4GL screen, by default.

Inserting and Deleting the Records

The records can be inserted and deleted within the screen array by means of the following keys:

Key	Action
F1	This key is used to insert a new screen record to the screen array below the row at which the cursor is currently situated. The values in the records below the new row are moved one row down and the cursor is moved to the beginning of the first field of the new record. This key is not required if the cursor is at the last row of the screen array. The user cannot insert a row if the number of the records of the screen array coincides with the number of records in the program array. If the user tries to do so, he will receive an error message informing that the program array is full. F1 is the default Insert key, it can be changed by the INSERT KEY option in the OPTIONS statement
F2	This key is used to delete a screen record at which the cursor is positioned from the screen array. The values in the rows below are moved up to fill the gap. F2 is the default Delete key, it can be changed by the DELETE KEY option in the OPTIONS statement

To make the corresponding changes to the program array the user must press the Accept key.

Large Data Types Used in the INPUT ARRAY Statement

The large data types are displayed by the 4GL in the following way:

- TEXT values – 4GL can display as much of the TEXT value as the form field can contain
- BYTE values – 4GL cannot display these values in 4GL field, the <BYTE value> character string will be displayed in the form field instead.

If the PROGRAM attribute is specified in the form specification file for the fields of BYTE or TYPE data types, the user can view the values of these fields by means of pressing the exclamation mark key (!), when the cursor is positioned in such field. The values will be viewed through an external editor, which allows the user to edit text or graphics.

When the external editor is opened, all the keys specified in the ON KEY or ON ACTION clauses are ignored. After the external editor is closed, 4GL restores the 4GL screen to the state in which it has been before the



external editor has been opened, resumes input at the same field, and reactivates the ON KEY and ON ACTION clauses.

Completing the INPUT ARRAY Statement

The INPUT ARRAY statement can be terminated by means of pressing:

- The Accept key
- The Interrupt key
- The Quit key
- The RETURN or TAB key when the cursor is at the last field of the form (if the INPUT WRAP option is **not** in the effect)

It will also be terminated, if 4GL executes the EXIT INPUT statement.

Pressing either of the above listed keys deactivates the form. The Quit and Interrupt keys also terminate the program unless the DEFER INPUT or DEFER INTERRUPT statement is in effect.

The user must press the Accept key to finish the input if:

- The INPUT WRAP option is in effect
- The AFTER FIELD clause for the last field contains the NEXT FIELD keywords

If a DEFER INTERRUPT statement has been executed previously and the user presses the Interrupt key, the following actions will be undertaken by 4GL:

- The global variable int_flag will be set to TRUE
- The INPUT ARRAY statement will be terminated but the program will be still running

If a DEFER QUIT statement has been executed previously and the user presses the Quit key, the following actions will be undertaken by 4GL:

- The global variable quit_flag will be set to TRUE
- The INPUT ARRAY statement will be terminated but the program will be still running

The control clauses are executed in the following order when the INPUT ARRAY statement is terminated:

1. AFTER FIELD clause (if the current field has this clause)
2. AFTER ROW clause
3. AFTER INPUT clause

If any of these clauses contains the NEXT FILED keywords, the INPUT ARRAY statement is not terminated, the cursor is moved to the specified field and the control is passed to the user.

If the EXIT INPUT statement is executed, none of the above listed clauses is executed.



LABEL

The LABEL statement serves as a marker for the GOTO statement, and marks the statement to be executed after 4GL passes control to it from the corresponding GOTO statement.



Element	Description
Label-name	The name of a label

The LABEL statement indicates the place to which the program control will be passed after the corresponding GOTO statement is executed. The GOTO and the LABEL statement must be in the same program block. For more information see the "[GOTO](#)" statement section.

There are some restrictions applied to the LABEL statement:

- The label-name must be unique within the program block
- The same label-name must be specified in the LABEL and GOTO statements to perform control transfer
- Both GOTO and LABEL statements must be within the same program block (MAIN, FUNCTION or REPORT block)

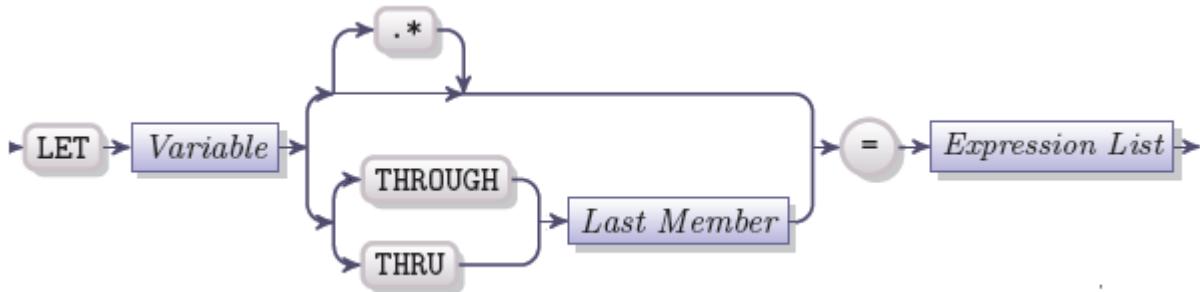
The label-name must be followed by the column symbol (:) unlike the GOTO statement where the column symbol is optional.

```
WHENEVER ERROR GOTO first_error
...
LABEL first_error:
ERROR "Incorrect format"
```



LET

The LET statement is used to assign a value or a set of values to a variable or a record.



Element	Description
Variable	A variable which is assigned a value
Expression List	One or more 4GL expressions that return the value which is assigned to the variable
Last Member	The last record member of the specified set of members (if the "variable" represents the first member of the sequence)

A variable of any data type except for the object data types (e.g. WINDOW, FORM) can be assigned values with the help of the LET statement.

When a variable is declared with the help of the DEFINE statement, it has no value, but the memory space is allocated to it. There are two ways of initializing a variable (that is of assigning some value to it): the use of the INITIALIZE statement or the LET statement. Do not use a variable that has not been initialized in a 4GL expression, the value returned by such expression will hardly prove to be useful.

When 4GL encounters a LET statement, it evaluates the expression to the right of the equal sign (=) and then assigns it to the variable. The following example illustrates assignment of values to variables:

```

DEFINE a, b, c, d INT
LET a=2
LET b=5
LET c=a+b
LET d=NULL
  
```

Most of the 4GL built-in functions and operators can be used in the LET statement. The example below assigns an ASCII symbol to the variable. The displayed value will be "d":

```

DEFINE var1 CHAR(1)
LET var1 = ASCII 100
DISPLAY var1
  
```



You cannot assign individual values to a program record with the help of the LET statement, you cannot use the THRU/THROUGH keyword as well. However, you can assign values of one program record to another program record with the help of the .* notation, provided that the both records have the same number of members and the members are of compatible data types:

```
LET rec1.* = rec2.*
```

The LET statement can be used to assign a value to a substring of CHAR, STRING or VARCHAR data type:

```
DEFINE full_name CHAR (30), fname CHAR (15)  
LET full_name[1,15] = fname
```

The LET statement can assign only NULL values to the variables of the Large data types. To assign other values to such variables use the INTO clause in a SELECT, FOREACH, OPEN, or FETCH statement or pass the name of such variable as an argument to a function.

The Coma and Double Pipe Symbols

The coma symbol (,) placed between the values located on the left of the equal sign (=) concatenates these values. A NULL value in such list of values separated by commas has no effect unlike in the list members of which are separated by the double pipe symbols (||). However, if every value in the list with coma (,) separator is NULL the whole expression is evaluated as NULL, whereas a list with the double pipe operator (||) is evaluated as NULL, if at least one its member is NULL.

```
DEFINE a, b, c CHAR(5),  
      d, e CHAR(15)  
LET a = "test1"  
LET b = "test2"  
LET c = NULL  
LET d = a, c, b  
LET e = a || c || b  
DISPLAY d  
DISPLAY e
```

The "d" value in the example above will be displayed as "test1test2". You will not see the value of the "e" variable, because its value will be NULL.

When you use these operators, take into consideration the effect of the NULL values.

In some locales the values of MONEY data type acquire separators and currency symbols specific to these locales and they may not coincide with the default separators and currency symbols. It is not influenced by the fact whether you use the USING operator in the LET statement or not.

Assigning Values to Array Elements

The LET statement can also be used to assign values to program array elements by listing the values after it. Such a way of value assignment may prove useful when there is no logical connection between the values of the elements (otherwise, the usage of conditional loops is more preferable).



This method is only applicable to one-dimensional arrays. If applied, all the array elements in the same LET statement should be listed, otherwise an error will occur. The values of the elements should be separated by commas:

```
DEFINE my_arr1 [3] OF INTEGER
LET my_arr1 = 11, 56, 23654
```

In the case of multidimensional arrays (two or three dimensions) the values should be assigned separately for each element of an array. Please, consider the following example:

```
My_arr1 ARRAY [2,3] OF CHAR(15)

LET my_arr[1,1] = "One-one "
LET my_arr[1,2] = "One-two "
LET my_arr[1,3] = "One-three"
LET my_arr[2,1] = "Two-one"
LET my_arr[2,2] = "Two-Two"
LET my_arr[2,3] = "Two-three"
```

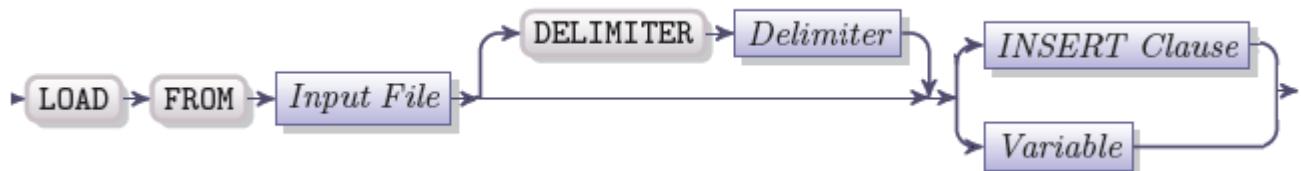
If you have an ARRAY of RECORD structure, you should keep to the following scheme of value assignment:

```
LET array_name[coordinate].record_member = value
LET rec_array[1].mem_1 = "Record value"
```



LOAD

To insert the data stored in a file into a database table, the LOAD statement is used.



Element	Description
Input File	A character expression that returns the name of the input file and its relative path if necessary
Delimiter	is a quoted string (or a CHAR, STRING, or VARCHAR variable) containing a delimiter symbol
Variable	A CHAR, STRING or VARCHAR variable containing an INSERT clause.
INSERT Clause	A CHAR, STRING, or VARCHAR variable containing an INSERT clause.

The LOAD statement must contain the INSERT clause to indicate the table to which the data should be loaded. The LOAD statement adds new rows to the specified table or view, it does not erase or delete the data already existing in this table or view. The LOAD statement cannot be used in the PREPARE statement. A row that has the same key as an existing row cannot be added to a table.

To execute the LOAD statement the user must have the insert privileges for the specified table.

The Input File

The name of the file from which the input will be performed should follow the LOAD FROM keywords. The file must contain the ASCII characters that represent the values that are to be inserted into a table.

The *file-name* in the LOAD FROM *file-name* clause can be represented by a variable of character data type which contains the name of the file. It can also be represented by the actual name of the file with the relative path to the file if necessary. The name and the path should be enclosed in the quotation marks. Below is an example of using a variable for specifying an input file.

```

DEFINE load_file CHAR(15)
LET load_file = "load_f.unl"
LOAD FROM load_file INSERT INTO my_table
  
```

The character string representation of the loaded values in the input file depends on the data type of the receiving table column.

Data Type of the Receiving Column	Format of the Input
CHAR / VARCHAR / TEXT / STRING	If the number of characters in the inserted value exceeds the declared size of the column, the excessive characters are truncated from the right.



	<p>If the value includes a literal backslash or other delimiter character, it should be preceded with the backslash symbol (\). The backslash (\) is also required before the NEWLINE symbol in a value of VARCHAR data type and as the last symbol of a TEXT value.</p> <p>Empty values can be represented by whitespace characters, however, the leading blanks must not precede the valued of the CHAR, STRING, VARCHAR, or TEXT data types.</p>
DATE	The format of a DATE value must correspond to the locale settings. According to the default locale, a DATE value must have the format mm/dd/yyyy. The default format can be changed by means of the DBDATE or GL_DATE environment variables. If you enter a year as the two digit number, the rest of the year number will be retrieved from the DBCENTURY variable. The values must be existent; you cannot enter June 32 or February 30.
DATETIME / INTERVAL	The INTERVAL value must be either of the year-month type (yyyy-mm) or day-fraction type (dd hh:mm:ss.fff), or any subset of these types. The DATETIME values must be in yyyy-mm-dd hh:mm:ss.fff format or a continuous subset of these values.
SERIAL	To make the database supply a new serial value to the inserted row, you can insert zero value (0) into the column of the SERIAL data type. If the column has a unique index and if you supply a number greater than zero that duplicates the existing serial number, an error will occur.
MONEY	The currency symbols can be used in the values but they are not required.
BYTE	The BYTE values must be ASCII-hexadecimals. Leading or trailing blanks are not allowed.

A set of values contained by the input file that represents a new row is called an input record. Each input record must be terminated with the help of a NEWLINE character. The NULL values must be represented by consecutive delimiters, nothing should be included between these delimiter symbols. The default delimiter symbol is the pipe symbol (|). Here is an example of the input records in an input file which inserts information in the columns that contain the serial number of the record, the last name, the first name, the gender, the phone number and the country:

```
0 | Smith | John | Male | | Canada
0 | Brown | Anna | Felame | (432) 231-65-30 | |
```

In the example above the NULL values are inserted into the column that should contain a phone number in the first record and a country name in the second one, two pipe symbols represent these values. If these input records are placed in a hypothetical input file called *ins_info*, the corresponding LOAD statement that inserts the information into *cust_info* table will have the following structure:

```
LOAD FROM "ins_info" INSERT INTO cust_info
```

The table name can be preceded by table qualifiers (i.e. by the owner identifier of the table). The number of the delimited values in the input records must be the same. If the INSERT clause contains only the name of the table, the input records must correspond to the columns in such table in number order and data types. The INSERT clause can contain the list of the columns which should match the input record values.



The file created by the UNLOAD statement can afterwards be used to populate another table with the help of the LOAD statement if the structure of this table is the same.

The LOAD statement expects the values to be supplied in the formats that correspond to the settings of the DBFORMAT, DBMONEY, DBDATE, GL_DATE, and GL_DATETIME environment variables. In case of discrepancy an error will occur and the execution of the LOAD statement will be cancelled.

The DELIMITER Clause

The default delimiter symbol is pipe (|). This default setting can be changed by means of the DELIMITER clause if the LOAD statement. The DELIMITER clause is optional. The delimiter sign must terminate an input record only if the last value of this record is NULL.

Below is an example of the LOAD statement with the DELIMITER clause which sets the caret (^) symbol as the delimiter:

```
LOAD FROM "/source/unl/ins_info" DELIMITER "^" INSERT INTO cust_info
```

The *ins_info* file will contain such input records:

```
0^Smith^John^Male^^Canada
0^Brown^Anna^Felame^ (432) 231-65-30^^
```

If the DELIMITER clause is omitted, the delimiter symbol is defined by the DBDELIMITER environment variable, if it is set, otherwise the default delimiter symbol pipe (|) is used.

The following symbols cannot serve as delimiters:

- Hexadecimal numbers (0 through 9, a through f, or A through F)
- NEWLINE (CONTROL-J)
- Backslash symbol (\)

The backslash (\) serves as an escape character that indicates that the character that follows the backslash should be interpreted literally and not as a delimiter symbol or escape character. If a character value includes the NEWLINE symbol or a delimiter symbol without backslash symbol, an error will occur. It is advisable that you use the LOAD statement within a transaction if your database supports explicit transactions, otherwise it would be difficult to recover the database after such error.

A backslash is automatically inserted by the UNLOAD statement before any literal delimiter or a literal NEWLINE symbol when the data are copied from a table into a file. These backslashes are removed by the LOAD statement, when the data are inserted into a table.

The INSERT Clause

The INSERT clause contains the identifier of a table or tables into which the values should be inserted. This clause supports the syntax of the independent SQL INSERT statement, however it is not possible to include optional clauses of the independent INSERT statement (i.e. VALUES, SELECT, EXECUTE PROCEDURE clauses) into the INSERT clause of the LOAD statement.

The INSERT INTO keywords can be followed by the table name (with or without table qualifiers) if the values in the input file match the columns of such table in order, number and data types.



You must explicitly specify the column names within the parentheses, if:

- The values are inserted not in every table column
- The order of values does not match the order of columns

```
LOAD FROM "/source/unl/ord_info" DELIMITER "!" INSERT INTO
.j_smith.order_info (ord_num, items, number)
```

The Data Integrity

If the LOAD statement is executed within a transaction, the rows into which the values are inserted are locked until 4GL encounters the COMMIT WORK or ROLLBACK WORK statement. If no other user is accessing the table, the whole table can be locked with the help of the LOCK TABLE statement to avoid locking limits. The lock is released when the transaction is finished.

The limit of locks available during a single transaction depends on your database server.

It is advisable that you use the BEGIN WORK statement and to include the LOAD statement in the transaction, if it is not an ANSI-compliant database but if it supports transactions. Otherwise it might be difficult to restore the database, if the insertion of one or more rows by means of the LOAD statement fails.

If an error occurs during the execution of the LOAD statement and the database does not support transactions, the rows that have been inserted by the LOAD statement before the error are not removed automatically. You need to remove them manually.

If the database supports transactions you can do the following:

- Run the LOAD statement as a singleton transaction (if an error occurs, the changes will be rolled back automatically)
- Run the LOAD statement within an explicit transaction (if an error occurs, the execution of the LOAD statement will be stopped but the transaction will remain open)

The transaction is not finished automatically after the LOAD statement is executed, unless the LOAD statement is followed by another LOAD statement. In an ANSI-compliant database (where a transaction is always in effect) the LOAD statement is the first statement of a new transaction if it is preceded by the DATABASE, COMMIT WORK or ROLLBACK statement.

In a non-ANSI-compliant database that supports transactions a new transaction is started by means of the BEGIN WORK statement. The LOAD statement is the first statement of a transaction only if it is immediately preceded by the BEGIN WORK statement.

If the LOAD statement is the first statement of a database transaction, this transaction is automatically committed when the execution of the LOAD statement is completed without errors.

If the LOAD statement is not the first statement of the transaction, the transaction is not committed and is left open, you can do the following:

- commit the rows that have been successfully inserted by means of the COMMIT statement, fix the loaded records and return LOAD with the balance of the records inserted
- abort the LOAD statement by executing the ROLLBACK WORK statement and begin the LOAD option from the very beginning



Performance Issues

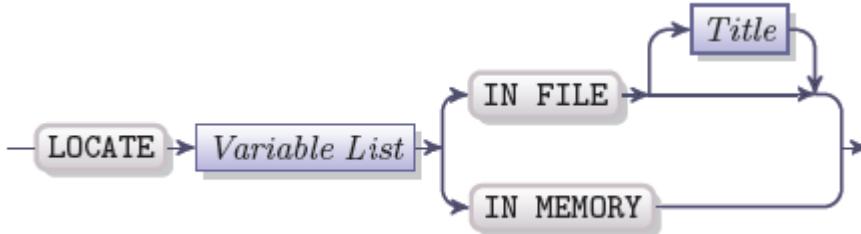
The performance is better, if the INSERT clause does not have an index, a constraint, or a trigger. If the table has an index, a constraint, or a trigger follow these steps to improve the performance:

1. Use SET INDEX...DISABLED statement to disable the index created on the table, if any
2. Use SET CONSTRAINT...DISABLED statement to disable the constraint created on the table, if any
3. Use SET TRIGGER...DISABLED statement to disable the trigger, if any
4. Use LOAD statement to insert data into the table
5. Use SET INDEX... ENABLED statement to restore the index created on the table, if any
6. Use SET CONSTRAINT...ENABLED statement to enable the constraint created on the table, if any
7. Use SET TRIGGER...ENABLED statement to enable the trigger, if any



LOCATE

The LOCATE statement specifies the location where the values for the variables of large data types are stored. It is the initialization method used for the variables of the large data types.



Element	Description
Variable List	A list of variables of large data types (TEXT or BYTE)
Title	A character expression returning a valid file name and optional path

The LOCATE statement is used to specify the location where you want to store the values for the TEXT or BYTE variables. These values can be stored in the memory or in a file. The values stored in the memory can be accessed faster. However, if your program exceeds the storage space allocated for it in the memory, a part of the value of large data type will be stored in a file. The variables of the large data types can be used in a 4GL program in the following way:

1. Declare a variable of the large data type with the help of the DEFINE statement
2. Use the LOCATE statement to specify the storage location for this variable. The LOCATE statement must be used within the scope of reference of the variable.

The LOCATE statement must follow any DEFINE statement that declares a variable of the large data type. If the variable is local, the LOCATE statement must occur within the same program block. If you try to use a variable of the large data type which has not been initialized with the help of the LOCATE statement, an error will occur.

Variables of the Large Data Types

The variables of the large data types in the LOAD statement can be specified in the following form:

- as a variable
- as a *record.**
- as a *record.first THRU record.last*
- as an array

If the variable has any prefixes specifying a record, an array, or other qualifiers, the corresponding record, array, etc. must be declared before the usage of the LOCATE statement by means of the DEFINE statement.

After a variable has been initialized, you can use most of the 4GL statements to access it. It can be assigned NULL value by means of the LET statement, but this statement cannot assign non-NUL values to the



variables of TEXT or BYTE data types. The INTO clauses of the SELECT statement can assign values of the database columns of the large data types to such variables.

The IN MEMORY Clause

The IN MEMORY clause indicates that storage in memory should be allocated for the values of TEXT or BYTE data type. The example below allocates memory for the variable *my_text*:

```
DEFINE my_text TEXT
LOCATE my_text IN MEMORY
```

If a variable of large data type has already been stored in memory, the LOCATE statement will reinitialize it. This statement also reinitializes a TEXT or BYTE variable stored in a temporary file. It is not possible to reinitialize a variable of large data type that is stored in a named file.

The IN FILE Clause

Use the IN FILE clause to store a TEXT or BYTE value in a named file. The name of the file is represented by a character expression that follows the IN FILE keywords, it can be:

- A quoted character string that contains the name of the file, its extention and relative path to it
- A variable of CHAR, VARCHAR or STRING data type
- A record member of CHAR, VARCHAR or STRING data type

The file is opened and closed each time you use the variable stored in an SQL or 4GL statement.

If you retrieve the value from a database column into this variable, the corresponding section of the file is overwritten. When a row is updated with the help of this variable, the entire file is read and stored in the specified column.

A variable of large data type stored either in memory or in a named file can contain only the most recent value assigned to it.

If the *file-name* is omitted, but the IN FILE clause is specified, the value will be stored in a temporary file.

The Temporary File

If the *file-name* is omitted after the IN FILE keywords, the value will be stored in a temporary file. This file will be created in the directory defined by the DBTEMP environment variable, when the 4GL program is run. If this variable is not set, the temporary file is located in the /tmp directory. If this temporary directory does not exist, 4GL will produce a runtime error.

```
DEFINE my_image BYTE
LOCATE my_image IN FILE
```

If the variable has been located in a temporary file, you may use the LOCATE statement to relocate it.

The Named File

To place the values of BYTE or TEXT data types in a specific file, the IN FILE keywords must be followed by the *file-name*. It can be either literal name of the file or a variable of character data type that contains the



name of the file. Optionally, it can also contain the relative path to the file and its extension. Here is the example of the literal representation of a file with its path:

```
DEFINE my_image BYTE
LOCATE my_image IN FILE "/source/img/my_image"
```

Below is an example of a variable used as the *file-name*:

```
DEFINE
    my_image BYTE,
    file_name VARCHAR(15)
LET file_name = "/source/img/my_image"
LOCATE my_image IN FILE file_name
```

Multiple file names can be specified by means of declaring an array of CHAR, VARCHAR or STRING data type:

```
DEFINE
    file_name ARRAY[5] OF CHAR(25),
    my_list ARRAY[5] OF TEXT
FOR x = 1 TO 5
    LET file_name[x] = "/source/list_files/list", x
    LOCATE my_list[x] IN FILE file_name[x]
END FOR
```

Passing Variables of the BYTE and TEXT Data Types to Functions

The arguments specified in the argument list of a function are usually passed by value. The function is able to modify the value passed in such a way without influencing the corresponding variable used in the calling routine.

Large data types are not passed by value, they are passed by reference. If a function changes the value of a large variable, it influences the variable used in the calling routine. You should not include the RETURNING clause into the CALL statement to return a value of the large data type.

Deallocating Memory

If the variables of the large data types are no longer needed, you can free the storage that has been allocated to them. To do it you can use the FREE statement – if the value of a large variable has been stored in a named file, referencing this variable with the FREE statement will delete this file. If the value has been stored in the memory, this statement frees the memory.

The local TEXT or BYTE variables of a REPORT or FUNCTION program block that are stored IN MEMORY or in temporary files are freed, when 4GL encounters the RETURN clause. 4GL does not deallocate the storage space used for the variables that are passed to a function as arguments. The storage for the non-local variables of TEXT and BYTE data types is deallocated when 4GL encounters the EXIT PROGRAM or END



MAIN keywords. If the value of a large variable is stored in a named file, this file is not deleted automatically if the FREE statement is not used explicitly.

You cannot use a variable of large data type which you have released. To be able to use this variable again, you must use the LOCATE statement again. If you want to use the file in which the value of large data type is stored, do not use the FREE statement.



MAIN

The MAIN statement indicates the beginning of the MAIN program block.



Element	Description
Program Block	A number of statements

An executable 4GL program can have no more and no less than one MAIN program block. It is usually used to call the functions and reports that are located within the same program module or in other program modules and libraries. The example below illustrates how the functions within the same program module are called in the MAIN program block:

```
MAIN
...
CALL first_finction()
CALL second_function()
...
END MAIN
FUNCTION first_finction()
...
END FUNCTION
FUNCTION second_function()
...
END FUNCTION
```

The MAIN statement can be used only as the first statement of the MAIN program block. It cannot be used in another 4GL or SQL statement. The END MAIN keywords are required, they mark the end of the MAIN program block. The program is terminated, when 4GL encounters the END MAIN program block. The program can terminate before the END MAIN keywords, if 4GL encounters the EXIT PROGRAM statement. The MAIN program block can include any 4GL statements and supported SQL statements. Typically it includes the DEFINE statement which declares the variables local to this program block. It can also include the DATABASE statement specifying the current database and the DEFER statement preventing the application from termination.

The Declaration of Variables

The variables declared by means of the DEFINE statement within the MAIN block are local to this MAIN program block, they cannot be referenced in other program blocks like FUNCTION or REPORT.



The variables declared before the MAIN keyword are module variables and can be referenced in the MAIN program block and in all the program blocks within this program module. The variables in the GLOBALS file have the scope of reference that is larger than one module. If you define variables with the same name as local, module and global variables, they will be used according to such precedence:

- A local variable will be used instead of any other variable of the larger scope of reference (module or global) within its program module
- A module variable will be used instead if a global variable with the same name within its program module

The DEFER and DATABASE Statements

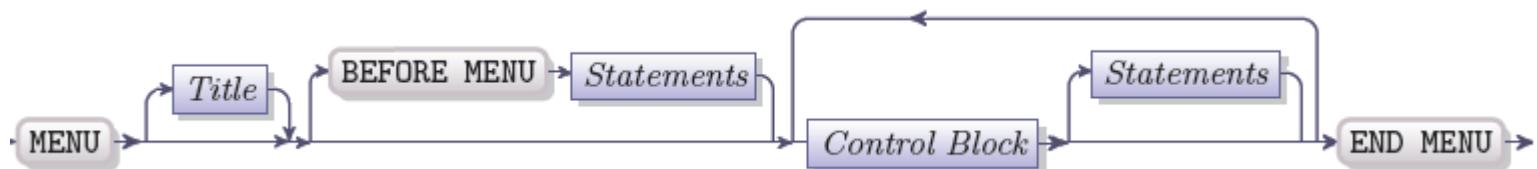
The DEFER statement can occur only within the MAIN program block.

The DATABASE statement that is located before the MAIN program block specifies the default database. This database becomes the current database if no other DATABASE statement is present within the MAIN block. A DATABASE statement within the MAIN block specifies the current database, this statement must occur after the DEFINE statement of the MAIN program block, if it is present. For more details about the current and default databases see the "[DATABASE](#)" statement section.



MENU

The menu statement provides your 4GL program with a ring menu. The user can choose menu options from the keyboard or with the help of the mouse and execute the corresponding blocks of statements.



Element	Description
Title	A character expression that returns a character string serving as the title of the menu
Menu Control Block	Block that contains menu commands
Statements	Executable statements and special keywords which are executed when the corresponding menu command is activated

The MENU statement specifies and displays the ring menu which occupies the first two lines of the 4GL screen: the first line contains the menu buttons and the second line displays the description of the currently highlighted menu option.

The MENU statement can be used to create and display the menu together with its title and the menu buttons that trigger menu options, specify a help message and description for each menu option.

4GL performs the following actions when it encounters the MENU statement:

1. The menu title is displayed together with the menu options which fit the length of the first menu line (the options that do not fit the screen can be viewed by pressing arrow buttons on both sides of the first menu line, or by means of LEFT and RIGHT arrow keys on the keyboard).
2. The cursor is moved to the first menu option and the description message for this option is displayed, if any.
3. 4GL waits until the user presses the menu button or an activation key or terminates the MENU by means of the Quit key or Interrupt key.
4. If the activation key is pressed, the statements of the corresponding Control Block are executed.

A Statements clauses can contain:

- The EXIT MENU statement which terminates the MENU statement
- The CONTINUE MENU statement which skips the remaining statements within the corresponding COMMAND clause and redisplays the menu.
- The HIDE OPTION/SHOW OPTION – specifies a COMMAND clause that does not contain an option name, its description or help number, but contains the statements and specifies the activation keys for their execution
- Any other valid 4GL statement.

When 4GL executes the last statement of a Statement clause, it redisplays the menu.



A menu cannot occur within a screen form, it should be located either above or below it. The menu is positioned in the MENU line, which is by default the first line of the 4GL screen or a 4GL window. This setting can be changed with the help of the MENU LINE option of the closest OPTIONS statement or OPEN WINDOW statement.

A runtime error occurs unless the 4GL screen can contain at least the menu title and one menu option.

The Menu Title

The menu title is a character expression (that can be a character variable or a quoted character string) that returns the title of the menu. This is a static display label, so it cannot be referenced by the program. If you want to use the same menu in different 4GL windows, you should write the MENU statement in a FUNCTION block and call this function when necessary.

The menu title is displayed when a program is run in the character mode. When the program is run with the help of a thin-client (e.g. Phoenix, Chimera, LyciaDesktop, or LyciaWeb), the menu title is not visible. It is hidden to make the client window look more like a standard application window. You can omit the title altogether.

The BEFORE MENU Clause

The statements in the BEFORE MENU clause are executed before the menu is displayed. This clause is usually included to perform the following tasks:

- To assign values to the variables used as the menu title, option names, descriptions, etc.
- To hide or show specified menu options
- To verify whether the user has the required privileges

If 4GL encounters the EXIT MENU keywords in the BEFORE MENU clause, the MENU statement is terminated without being displayed. In the example below the BEFORE MENU clause uses the *admin_operator* function to check whether the current user is the admin and if not hides a menu option:

```
MENU "Main"  
BEFORE MENU  
    IF NOT admin_operator(g_operator.name) THEN  
        HIDE OPTION main_menu[6].option name  
    END IF  
    ...  
END MENU
```

The Menu Control Block

A Menu Control Block can be represented by one of the following three options:

- COMMAND clause
- ON ACTION clause
- ON IDLE clause

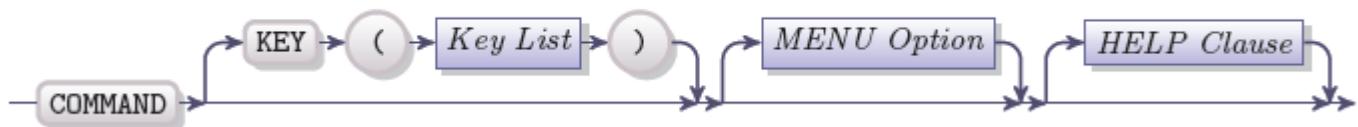
the COMMAND and ON ACTION clauses can appear within the MENU statement in any order and their number is not limited. The ON IDLE clause can appear only once in each MENU statement.



The MENU statement is displayed on the screen as a ring menu of option names. The menu options are placed in the order in which the COMMAND or ON ACTION clauses occur within the MENU statement. A menu must contain at least one non-hidden option. The description of a highlighted menu option, if it is set, appears below the first menu line.

The COMMAND Clause

The COMMAND clause is used to specify the name, key, description and help number for a menu option. It also contains the statements that are executed when this option is activated.

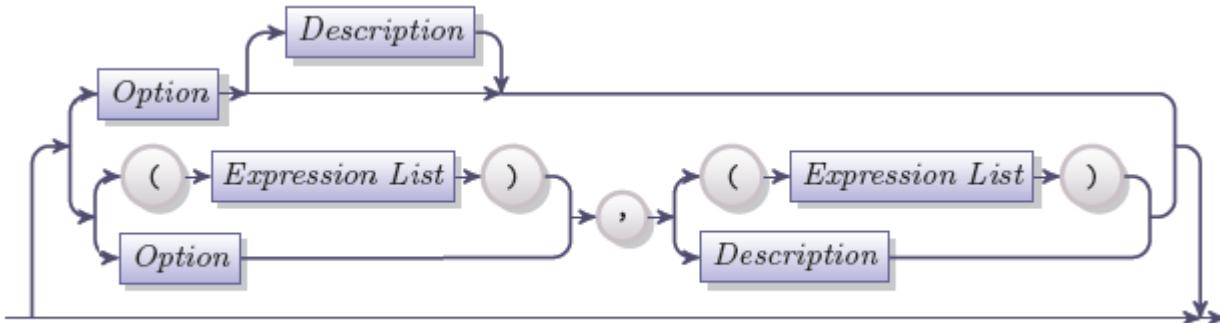


Element	Description
Key List	A list of one or more keys that invoke the menu option
MENU Option	The clause that contains the name and description of a menu option
HELP Clause	The clause which contains help number corresponding to the menu option

A COMMAND clause can contain a nested MENU statement.

The Menu Option

The menu option clause contains the name of the menu option and the description. Both are optional. This clause has the following syntax:



Element	Description
Option	A character variable or a character string used as the name of a menu option
Description	A character variable or a character string used as the description of a menu option
Expression List	A 4GL expression that returns a name or a description of a menu option



There are two ways of specifying the option name and the description:

- In the first case where we have no separator between the option name and description
 - Both name and description in this case are optional
 - If only one of them is present, it is treated by 4GL as the option name.
 - Both option name and description can be represented either by a quoted character string (or other constant), or by a variable.
 - Typically, the variable used is of character data type, though it can be of any other simple or structured data type, provided that it makes sense as a menu option name or description.
- In the second case a comma separator is present between the name and description
 - Both name and description in this case are obligatory
 - If only one of them is present a compile-time error occurs
 - Both name and description can be represented by a variable, a constant, or a 4GL expression
 - If a name or a description is represented by a 4GL expression which consists of more than one variable or has any operators, it should be enclosed in parentheses, e.g.:

```
MENU "my_menu"  
COMMAND KEY (F1) (curr_user||status), "Press to change the user"
```

	Note: An error will occur if the description is longer than the line at which it is displayed in the character mode. If the application is run by a thin client, the excessive symbols are truncated from the right.
--	---

If a COMMAND clause contains no option name and no description, the menu option will not be visible for a user.

The HELP Clause

The HELP clause specifies the number of the help message associated with the corresponding menu option.



Element	Description
Integer Expression	A 4GL expression that returns a positive integer greater than 0 and less than 32,767.

The message will be displayed if the menu option is highlighted and the user presses the Help key. By default CONTROL-W is the Help key, this setting can be changed by means of the HELP KEY option of the closest OPTIONS statement.

```
MENU  
COMMAND "Add" "Add a new record" HELP 23  
CALL add_rec()  
...  
END MENU
```



In the example above, the help message number 23 is associated with the menu option "Add". The number must occur in the help file that is specified in the HELP FILE option of the OPTIONS statement.

A menu option that is invisible cannot have a HELP clause.

The KEY Clause

The KEY clause contains from one to four keys that activate the corresponding menu option. If the KEY clause is absent, the first character of the option name is the activation key for the menu option. If the KEY clause is present, the first letter of the option name cannot be used as the activation key. It is advisable that you use unique option names that begin with different letters, if you do not plan to specify the KEY clauses for them.

If the activation key is pressed, 4GL executes the statements contained in the COMMAND clause of the option. After it has executed the last statement of the clause, the menu is redisplayed, unless the COMMAND clause contains the EXIT MENU or EXIT PROGRAM keywords.

In the example below the first menu option is activated with the help of the CONTROL-P key, specified in the KEY clause, the second one is activated with the help of the F key – the initial letter of the option name, because the KEY clause is absent:

```
MENU "Main"  
  COMMAND KEY (CONTROL-P) "Add" "Add a new record"  
  ...  
  COMMAND "Find" "Search for a record"  
  ...  
END MENU
```

The following keys can be specified in a KEY clause:

- Letter keys (4GL makes no difference between upper and lower case letters)
- Symbols enclosed in quotation marks (i.e. !, @, #)
- Any of the following keywords: DOWN, ESC/ESCAPE, F1 through F256, INTERRUPT, LEFT, RETURN/ENTER, RIGHT, TAB, UP and the combination of the CONTROL key with any letter key (except for the letters A, D, H, I, J, K, L, M, R, and X).

Some keys require special attention if used in the ON KEY clause:

Key	Usage Features
ESC/ESCAPE	If you want to use this key in ON KEY block, you must specify another key as the Accept key in the OPTIONS block, because ESCAPE is the Accept key by default
INTERRUPT	DEFER INTERRUPT statement must be executed before you will be able to use this key in the KEY block. On pressing the Interrupt key, the corresponding COMMAND clause is executed, int_flag is set to non-zero but the MENU statement is not terminated
QUIT	DEFER QUIT statement must be executed before you will be able to use this key in the KEY block. On pressing the Quit key the corresponding COMMAND clause is executed and int_flag is set to non-zero but the MENU statement is not terminated
CTRL-char	4GL reserves these keys for field editing and they should not be used in the



**(A, D, H, K, L,
 R, X)** KEY block

**CTRL-char
 (I, J, M)** These key combinations by default mean TAB, NEWLINE and RETURN. If they are used in the KEY block, they cannot perform their default functions while the KEY block is functional.

You may not be able to use the keys S, Q and Z in a KEY clause on some versions of UNIX OS. You may not be allowed to use other keys which may be reserved by your operational system.

The Invisible Menu Options

An invisible option is an option that is not displayed. To add an invisible menu option, you should specify the KEY clause in the COMMAND clause, but the option name and option description should be omitted.

As well as in the visible menu options, the key specified in the KEY clause of the invisible menu options must be unique within the menu. If you choose a letter key as an activation key, it should differ from the names of the visible menu options.

In the example below the MENU statement has two options, the second option is invisible:

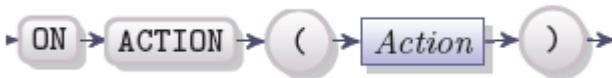
```
MENU
COMMAND "Update" "Update an existing record" HELP 90
    CALL upd(current_rec)
COMMAND KEY ("!")
    CALL upd_all()
END MENU
```

The "Update" is a visible option, it can be activated by pressing U key. The invisible option is activated when the user presses "!" key.

You must specify at least one visible option in a menu. The MENU statement can include a COMMAND clause in which the menu option is null, it is represented by empty quoted string.

ON ACTION Clause

The ON ACTION clause of a menu serves the same purpose as the COMMAND clause - it creates menu options. This clause gives you less flexibility in specifying the option name and description, but instead it allows you to make custom actions the menu option triggers. While the menu options defined by the COMMAND clause have the activation key, either default or specified in the KEY clause, the menu options specified by the ON ACTION clause have actions assigned to them. They cannot be triggered from the keyboard, but can be invoked using form widgets with the corresponding action assigned.
 The menu ON ACTION clause is very similar to the ON ACTION clause in the user interaction statements such as NPUT. However, it has some restrictions. Here



Key	Usage Features
Action	The name of the action with or without the quotation marks



The ON ACTION clause allows only one action to be assigned to the menu option. The action name can be with or without quotation marks. 4GL expressions are not allowed in the action specifications. The ON ACTION CLAUSE also does not have the optional HELP clause or the description for the menu option. The name of the menu option displayed will be the same as the action name used.

The ON ACTION and COMMAND clauses can be present in the same MENU statement, for example:

```
MENU
COMMAND "my_command" "Update an existing record"
    CALL upd(current_rec)
ON ACTION exit
    EXIT MENU
END MENU
```

ON IDLE Clause

The ON IDLE clause can appear only once per MENU statement. It specifies the amount of time the program should be idle for the statements included into the ON IDLE clause to be executed. A program is considered idle, if the user does not take any action - there is no key pressed or mouse clicks - during the specified period of time. Here is the clause syntax:



Key	Usage Features
Seconds	The integer value or an integer expression that returns the number of seconds

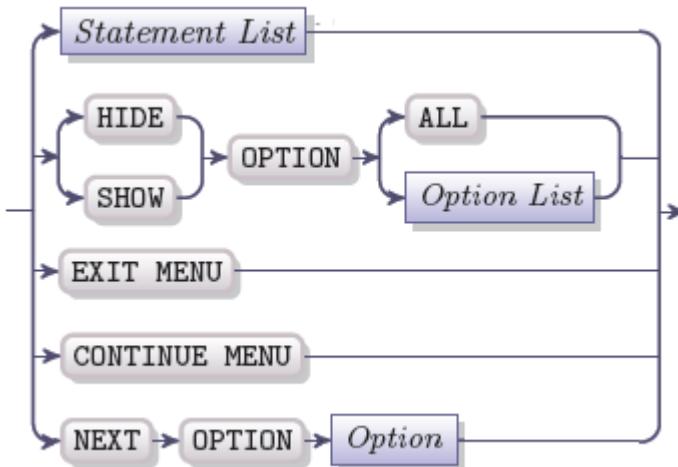
Once the program is idle for the specified number of seconds, the statements in the ON IDLE clause are executed. After the last statement is executed, the program redisplays the menu. The ON IDLE clause will be triggered as many times during the program execution, as often it will go idle. Do not specify too short idle period, otherwise the program execution will be constantly interrupted by the execution of the ON IDLE clause.

Here is a typical example of an ON IDLE clause:

```
MENU
...
ON IDLE
    CALL fgl_winmessage("The program was")
ON ACTION exit
    EXIT MENU
END MENU
```

The Statement Block

The statement block that follows a menu command block has the following structure:



Element	Description
Statement List	4GL or SQL executable statements
Option List	The list of one or more valid menu option names enclosed in quotation marks
Option	The name of the menu option you want to be highlighted as the next menu option enclosed in quotation marks

The CONTINUE MENU Keywords

When 4GL encounters the CONTINUE MENU keywords, it skips the rest of the statements of the MENU control clause and redisplays the menu. The user is free to choose another menu option.

```

MENU "Edit"
COMMAND "Update" "Update an existing record" HELP 90
PROMPT: "Are you sure you want to update the row?" FOR y_n
    IF y_n MATCHES "[Yy]" THEN
        CALL upd(current_rec)
    ELSE
        CONTINUE MENU
    END IF
    ...
END MENU
    
```

The EXIT MENU Keywords

When 4GL encounters the EXIT MENU statement, it terminates the menu and does not execute any statements between the EXIT MENU keywords and the END MENU keywords. The menu is not redisplayed. This option should be specified at least in one menu option, otherwise the user will be unable to leave the menu.



When the EXIT MENU statement is executed, the menu is deactivated and erased from the screen. 4GL executes the statements that follow the END MENU keywords.

```
MENU "Edit"  
    COMMAND "Update" "Update an existing record" HELP 90  
        CALL upd(current_rec)  
    COMMAND "Exit" "Leave the Edit menu"  
    EXIT MENU  
END MENU
```

The NEXT OPTION Keywords

The NEXT OPTION keywords are used to specify which menu option will be the current menu option when 4GL finishes execution of the currently executed COMMAND clause. If these keywords are not present in the COMMAND clause, the menu option that has just been executed remains the current menu option. The NEXT OPTION clause does not activate the menu option which is specified in it, it just highlights the menu option which is indicated as the next option after the current COMMAND clause is executed.

```
MENU "Contacts"  
    COMMAND "Find" "Search for contact information" HELP 11  
        CALL search()  
        NEXT OPTION "Edit"  
    COMMAND "Edit" "Edit contact information" HELP 12  
        CALL add_con()  
    ...  
END MENU
```

The above code sample specifies the "Edit" menu option as the next option, it will be highlighted after the "Find" option is executed.

By default, the cursor will move through the menu in the order in which the COMMAND clauses are listed. Use the NEXT OPTION keywords only if you want to deviate from the default order of the cursor movement, which is from left to right.

The HIDE OPTION and SHOW OPTION Keywords

The HIDE OPTION and SHOW OPTION keywords can be used to specify which of the menu options will be visible to user and which will be hidden. When the option is hidden with the help of the HIDE OPTION keywords, 4GL does not display this option in the menu and does not react to the keystrokes that should activate this option. The option remains hidden and disabled until the SHOW OPTION keywords are used with this option.

```
MENU "Contacts"  
    COMMAND "Find" "Search for contact information" HELP 11  
        CALL search()
```



```
NEXT OPTION "Edit"  
COMMAND "Edit" "Edit contact information" HELP 12  
    CALL add_con()  
COMMAND "Delete" "Delete the contact" HELP 13  
    CALL del_con()  
COMMAND "Demo menu" "Display only demo options"  
HIDE OPTION ALL  
SHOW OPTION "Find"  
COMMAND "Standard menu" "Display all options"  
SHOW OPTION ALL  
END MENU
```

If you use the variables of character data types as the option names, you must assign values to them before you can show or hide such options.

You can use the ALL keywords together with the HIDE OPTION or SHOW OPTION keywords to show or hide all menu options. By default all the options are displayed. To display the previously hidden options use the SHOW OPTION keywords.

The menu options are displayed in the same order in which the COMMAND clauses are specified in the source code. The order in which the options are listed in the HIDE OPTION or SHOW OPTION clauses has no effect on the order of their appearance or disappearance.

The hidden options must not be confused with invisible options. Invisible options are not displayed on the screen, however they cannot be hidden or shown with the help of the HIDE OPTION or SHOW OPTION keywords, because they do not have the option name by which the menu options are referenced. Thus the invisible options cannot be disabled by means of the HIDE OPTION keywords and they cannot be displayed. To disable an invisible menu option, you can use this option in a conditional statement.

The HIDE OPTION or SHOW OPTION keywords can also be used not only in the COMMAND clause but also in the BEFORE MENU clause.

The Nested Menus

One MENU statement can be nested in another MENU statement. The nested MENU statement can occur in a statement block within the other MENU statement, or in a function that is called from another MENU control clause.

The END MENU Keywords

The end of the MENU statement is marked with the END MENU keywords. These keywords must follow the last COMMAND clause. Every MENU statement requires these keywords. If a MENU statement is nested within another MENU statement, both statements should have their own END MENU keywords.

If 4GL encounters the EXIT MENU keywords, it executes the statements that follow the END MENU keywords. To terminate the currently executed MENU control clause but not to terminate the menu itself, use the CONTINUE MENU statement rather than END MENU or EXIT MENU keywords.



Variables Used as Menu Identifiers

A variable can be used to specify:

- The menu title
- The menu option name (in a COMMAND clause and in the NEXT OPTION, SHOW OPTION and HIDE OPTION clauses)
- The menu option description

The values must be assigned to such variables before they can be used in the MENU statement. The values can be assigned before the MENU statement or within the MENU statement in any MENU control clause.

If you change the value of a variable used as an identifier in the MENU statement, be aware of the following:

- The length of the menu title is a constant value and it is determined when 4GL first displays the menu. The new value for a variable must fit the determined length. If the new menu title is longer than the space provided, 4GL displays only those characters that fit the length and the excessive characters are truncated from the right. If the new title is shorter than the space provided, trailing blanks are added.
- If an element of an array is used as a variable within a MENU statement, the value of such variable will be calculated only once before the menu is displayed. To index into the array the value of the index variable is used after 4GL executed the BEFORE MENU clause. Any changes made in this variable afterwards do not influence the values displayed by the MENU option.

If the length of the variable value that is used in the MENU statement exceeds the screen length, an error occurs.

Activating a Menu Option

To activate the menu option, the user can:

- Use the arrow keys to position the cursor and press the RETURN key
- Type the key sequence specified in the corresponding KEY clause
- Type the first letter of the option name, if no KEY clause is specified for the option

Scrolling through the Menu Options

If the width of the screen does not allow 4GL display all the menu options at a time, they will be displayed partially with the ellipsis (...) at the right side or at the left side in the character mode, or with the arrow buttons at both sides of the menu line in GUI mode.

To scroll through the menu options in the character mode the user can use the following keys:

Keys	Effect
→ or SPACEBAR	The RIGHT ARROW or SPACEBAR key moves the cursor to the next option. If there is an ellipse at the right and the cursor is at the rightmost option of the screen, the next page of options is displayed. If there is no ellipse at the right and the cursor is at the rightmost option, it will be returned to the first menu option.
←	The LEFT ARROW key moves the cursor to the previous option. If there is an ellipse at the left and the cursor is at the leftmost option of the screen, the previous page of options is displayed. If there is no ellipse at the left and the cursor is at the leftmost option, it will be returned to the last menu option.



↑	The UP ARROW key moves the cursor to the first menu option of the previous page of options
↓	The DOWN ARROW key moves cursor to the first menu option of the next page of options

To scroll through the menu options in the GUI mode the user can use the mouse left button by which the user can press the menu buttons directly without having to scroll though the menu. To see pages with the options that did not fit the screen, press the arrow buttons located at the left and at the right of the menu line.

It is also possible to use keyboard to scroll though the menu options in the GUI mode, though the effect of the keys will be slightly different:

Keys	Effect
→	The cursor is moved to the next option. If the cursor is at the rightmost menu option of the screen, the next page of options is displayed, if the arrow buttons are present at both sides of the menu line. If the cursor is at the rightmost menu option of the current menu, it will be returned to the first menu option.
←	The cursor is moved to the previous option. If the cursor is at the leftmost menu option of the screen, the previous page of options is displayed, if the arrow buttons are present at both sides of the menu line. If the cursor is at the leftmost menu option of the current menu, it will be returned to the first menu option.
↑	The cursor is moved to the previous option. If the cursor is at the leftmost menu option of the screen, the previous page of options is displayed, if the arrow buttons are present at both sides of the menu line. If the cursor is at the leftmost menu option of the current menu, it will be returned to the first menu option.
↓	The cursor is moved to the next option. If the cursor is at the rightmost menu option of the screen, the next page of options is displayed, if the arrow buttons are present at both sides of the menu line. If the cursor is at the rightmost menu option of the current menu, it will be returned to the first menu option.

You cannot use the SPACEBAR key to scroll through the menu in the GUI mode.

The menu is disabled when the INPUT, CONSTRUCT, or INPUT ARRAY statements are being executed.

Completing the MENU Statement

To terminate the menu statement the user should press the Interrupt key. If 4GL encounters the EXIT MENU keywords, the MENU statement is also terminated.

If the DEFER INTERRUPT statement is in effect, pressing the Interrupt key will trigger the following actions:

- The int_flag built-in variable will be set to non-zero value
- The control remains in the MENU statement until the EXIT MENU keywords are encountered.

At least one EXIT MENU statement is required in every MENU statement. Usually it is placed in the COMMAND clause with the "Exit" or "Quit" menu option. If the menu is nested, the EXIT MENU terminates only the current menu and returns control to the enclosing MENU statement.



The Conflicts of the Activation Keys

The activation key is either the first letter of the menu option, or the key sequence specified in the KEY clause of the menu COMMAND.

First menu option Letter as the Activation key

If the user types a letter, 4GL verifies whether this letter is unique among the options names:

- If the letter is unique among the initial letters of the menu options, 4GL executes the menu option associated with the letter.
- If the letter is not unique and more than one option begins with it, the option on the right which is closer to the current position of the cursor will be activated. If the cursor is placed on one of non-unique options, this option will be activated when the activation key is pressed.

Activation Key in the KEY Clause

The used key in the KEY clause should be unique among other KEY clauses within the same menu. You can use the default key for a menu option as activation KEY in the menu clause, this will not cause an error. If the KEY clause is present, the initial letters of the menu options cease to be their activation keys and produce no effect when pressed.

If you specify the same key as the activation key in the KEY clauses of several menu options, the first option on the list will be activated if the user presses this key. The order of the menu options is determined by the order in which their COMMAND clauses are listed within the MENU statement.

```
MENU "Main"  
COMMAND KEY (F6, "U") "Update" HELP 87  
...  
COMMAND KEY (F6, "G") "Search" HELP 90  
...  
END MENU
```

If the user presses the "F6" key in the example above, the "Update" option will be activated, but the "Search" option will never be activated with the help of this key.

If you specify the activation key for one menu option the same as the initial letter of the other option without the KEY clause, the KEY clause will prevail over the initial letter as the default activation key. In the example below pressing "S" key will result in selecting the "Add" option rather than "Search" option:

```
MENU "Main"  
COMMAND KEY ("S") "Add"  
...  
COMMAND "Search"  
...  
END MENU
```

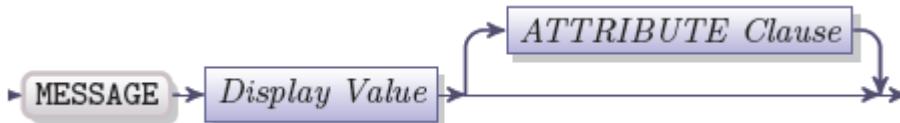


Even if the cursor will be at the "Search", pressing the "S" key will nevertheless activate the "Add" option.



MESSAGE

The MESSAGE statement specifies a message that is displayed on the reserved message line.



Element	Description
Display Value	The information which will be displayed by the MESSAGE statement
ATTRIBUTE Clause	Attributes that apply to the displayed information

The display value can be:

- A quoted character string which represents the text of the message.
- A variable or variables of CHAR, VARCHAR or STRING data types that contain the message text. If you use several variables, their values will be concatenated into one character string.
- A variable or variables of other data types (except BYTE data type).

The SPACE and CLIPPED operators can be used in the MESSAGE statement to organize the variables included into the display value.

```
DEFINE creat_date DATE,  
      file_name CHAR(30)  
  
...  
LET creat_date = 01/17/2010  
LET file_name = "my_list.txt"  
MESSAGE "The file ", file_name CLIPPED, 1 SPACE, "was created on ",  
creat_date
```

The above code will display the following message on the screen:

```
The file my_list.txt was created on 01/17/2010
```

The message remains on the screen until a menu is displayed, which uses the same line for being displayed, the erasing of the message can be avoided, if you specify another line as the message line. A message will be replaced by another message, if it is displayed.

If the message is longer than the message line, the message will be truncated from the right to fit the line.

The Message line

A message is displayed on the message line. By default, the message line is the second line of the 4GL screen. If the menu is displayed it prevents the messages from being displayed as it uses the second line for displaying descriptions for the menu options. Even if there are no descriptions for the menu options, this line



remains occupied by the MENU statement and MESSAGE text cannot be displayed if it uses the default message line.

Message text will also be invisible if the message line coincides with the form fields.

The default position of the message line can be changed by means of:

- The MESSAGE LINE option in the closest OPTIONS statement
- The MESSAGE LINE option in the ATTRIBUTE clause of the most recent OPEN WINDOW statement

To clear the message line you can display an empty character string:

```
MESSAGE " "
```

The ATTRIBUTE Clause

The general information about the ATTRIBUTE clause can be found in the [ATTRIBUTE clause](#) section of this reference. There are some specific issues that concern the usage of the ATTRIBUTE clause in the MESSAGE statement.

By default, the message text is displayed with the NORMAL attribute. To specify other attributes, use the ATTRIBUTE clause. The below example displays the MESSAGE text in green and with reversed video:

```
MESSAGE "This is a green reversed message" ATTRIBUTE (GREEN, REVERSE)
```

The INVISIBLE attribute is ignored in the MESSAGE ATTRIBUTE clause.

It is possible to use substrings of CHAR, VARCHAR, STRING or TEXT data types in the MESSAGE statement by specifying the starting and ending points of the substring in the square brackets:

```
MESSAGE "The goods descriptions were added to the table: ", descr [165,  
170] CLIPPED, 1 SPACE, descry [405, 420]
```



NEED

The NEED statement controls output from the PRINT statement. This statement can occur only in the REPORT program block.



Element	Description
Integer Expression	The integer expression which returns a positive integer specifying the number of free lines required

If the NEED statement is used in the PRINT statement, the printing report will start from the new page if the number of free lines at the bottom of the page is fewer than the number of lines specified in the NEED statement. The conditional statement with the SKIP TO TOP OF PAGE option has the same effect as the NEED statement. You can use either LINE or LINES keyword, they are synonyms.

Use the NEED statement to prevent parts of your report from being separated and to place relative parts on a single page.

```
AFTER GROUP OF rec.ord_id  
NEED 10 LINES  
PRINT " ",rec.ord_date, 4 SPACES,  
GROUP SUM(rec.ord_tot_price) USING "$$, $$, $$ . &&"
```

The value in the NEED statement does not include the BOTTOM MARGIN. If the number of lines left at the bottom of the page is less than the number specified in the NEED statement, the PAGE TRAILER and PAGE HEADER are printed before 4GL executes the following PRINT statement. The NEED statement cannot be included into PAGE TRAILER or PAGE HEADER clause.



OPEN FILE

The OPEN FILE statement is used to open a file containing some sort of text, usually separated by delimiters in order to read the information from it into the program variables, or to record the program variables values into it. Here is its syntax:



Element	Description
Descriptor	An integer variable that contains the file descriptor
Character Exp	A quoted character string or a variable of a character data type that contains the name of the file with the optional path
Option List	A list of options separated by commas that define the opening mode and other settings.

If the OPEN FILE statement was executed successfully, it stores the file descriptor as a positive integer value into the file descriptor variable. There may be no direct dependence between the file descriptor value returned by OPEN FILE and the system-level file descriptor.

Descriptor

The descriptor is an integer variable that contains a positive integer. It is important that all your open file sessions have different values stored in their descriptors.

Though the descriptors of the two opened files are different variables, they may have the same value stored. Typically it happens if a non-initialized variables are used as a descriptor - in this case all descriptors will store 0. Lycia will produce an error in this case, since the file descriptors values should be unique among all open files.

Options

The OPTIONS clause defines whether the file is opened in the write, read or append mode, what delimiter and what file format is used.

There is the list of possible options. Note that all the options may be present at the same time, e.g. read and write options are not mutually exclusive.

Option	Description
READ	Opens the file in read mode.
WRITE	Opens the file in write mode.
APPEND	Opens the file in append mode.
CREATE	Creates the file specified for opening, if the file does not already exist.
EXCLUSIVE	Only used together with the CREATE option. If the file you try to open already exists, the EXCLUSIVE option throws an error.



DELIMITER="delimiter" This option specifies the delimiter that is used in a file of the "load" format.

FORMAT="format" The format of the opened file. The format defines the default delimiters used, there are three possible options: "load", "csv", and "text"

READ

Files opened in the read mode can be read into the program variable. If there is no WRITE or APPEND option together with READ, the file contents cannot be changed. Its contents gets read according to the delimiters, default or custom. For more information about reading mode see the READ statement below in this guide.

WRITE

Files opened in the write mode can be recorded into the file from the program variables. Its contents gets written according to the delimiters, default or custom. While the file session is active, the WRITE statement records new values at the bottom of the file. Every new open file session with the write mode opens the file with the offset position at the beginning of the file, so the file contents gets overwritten.

```
OPEN FILE f_desc FROM "/temp/clients.txt"  
      OPTIONS(READ, WRITE, FORMAT = "text")
```

For more information about writing mode see the WRITE statement below in this guide.

APPEND

Files opened in the append mode can be recorded into the file from the program variables. Its contents gets written according to the delimiters, default or custom. Every new open file session with the append mode opens the file with the offset position at the end of the file, so the file contents does not get overwritten.

```
OPEN FILE f_desc FROM "pricelist.csv"  
      OPTIONS(APPEND, FORMAT = "csv")
```

For more information about writing mode see the WRITE statement below in this guide.

FORMAT

There are three formats a file can be opened in:

- text - this format means a standard text file, that is read and written by full lines. The default delimiter for this type of file is new line symbol (\n). You cannot assign any other delimiters using DELIMITER option - an error will occur.
- csv - this format means a csv file whose values are separated by the comma (,) delimiter. A different delimiter cannot be assigned for this file format.
- "load" - this format means a standard unl file or any other type of file whose contents is separated by the delimiters. The default delimiter is pipe (|). The default delimiter can be changed by the means of the DELIMITER option.



DELIMITER

The DELIMITER option can be used to set a custom delimiter to the files of the "load" format. By default the pipe (|) is the default delimiter for the "load" file format. If you intend to use the default delimiter, it can be omitted. If you try to specify any other delimiter than the default delimiter for a text (default delimiter is \n) or csv (default delimiter is ,) file, a compile time error will occur.

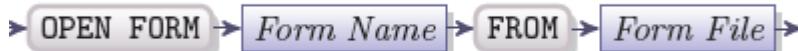
Here is an example of a custom delimiter for an unl file:

```
OPEN FILE f_desc FROM "config.unl"  
OPTIONS(READ,APPEND,CREATE, FORMAT = "load", DELIMITER="^")
```



OPEN FORM

The OPEN FORM statement declares a *form-name* of a compiled 4GL screen form so that it could be referenced by the DISPLAY FORM statement.



Element	Description
Form Name	The name you want to assign to the opened form
Form File	A 4GL expression that returns the name of the form file and its relative path, if required

To display a screen form, follow these steps:

1. Create a screen form (in the .per format or in the .4fm format) and compile it.
2. Declare the name of the form by means of the OPEN FORM statement
3. Display the declared form with the help of the DISPLAY FORM statement

The displayed form can be activated by means of the CONSTRUCT, DISPLAY ARRAY, INPUT, or INPUT ARRAY statement. When the OPEN FORM statement is executed, the form file is loaded into memory.

The Form Name

The form-name that follows the OPEN FORM keywords can be represented by a character string, it need not to be declared previously. The form-name does not need to match the name of the form file. However, it should be unique among the other forms used by the program. The form-name can be referenced in any part of the program once it has been declared by means of the OPEN FORM statement.

The file-name of the form must follow the FROM keywords. It must specify the name of the file without the extension, but with the relative path, if it is required. The file-name should be enclosed into the quotation marks. The path is required for form files located not in the same folder as the file which contains the corresponding OPEN FORM statement.

```
OPEN FORM my_form FROM "/forms/form_file1"
```

Form File

The form file can be specified in the following ways:

- A quoted string that contains the name of the file and its relative path if necessary
- A variable of character data type (CHAR, VARCHAR, or STRING), a record member or array element of the character data type that returns the name and path of a form file
- Any 4GL expression that returns the name and the path of a form file



The name of the form file **must not** include the file extension. Below is an example of a 4GL expression used after the FROM keyword:

```
DEFINE
    f_file STRING,
    id_number SMALLINT
LET id_number = 12
LET f_file = "file00"||id_number
OPEN FORM my_f FROM f_file
```

The form file opened in the example above must be called "file0012".

Displaying a Form in a 4GL Window

There are two ways of displaying a screen form in a 4GL window:

- Place the OPEN FORM and DISPLAY FORM statements after the OPEN WINDOW statement

```
MAIN
OPEN WINDOW window1 AT 2,2 WITH 15 ROWS, 50 COLUMNS
ATTRIBUTE (BORDER)
OPEN FORM my_form FROM "forms/form_file1"
DISPLAY FORM my_form
...
END MAIN
```



Note: If the size of the form displayed by the DISPLAY FORM statement is larger than the window in which it is displayed, a runtime error -1142 will occur, informing you that the window is too small to display the form

- Use the OPEN WINDOW ... WITH FORM statement

```
OPEN WINDOW window2 AT 2, 2 WITH FORM "form_file1"
```

When a form is displayed by means of the OPEN WINDOW...WITH FORM statement, the window will be the same size as the form, so you cannot receive error -1142.

If you use the same form-name in another OPEN FORM statement, the previously opened form will be closed and 4GL will open a new one.

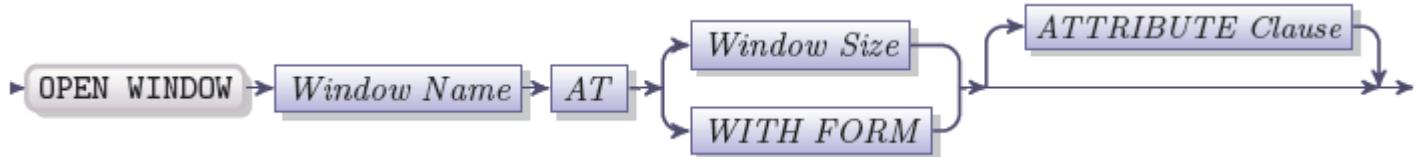


If you open a screen form in a 4GL window, you do not need to use the statement CLOSE FORM to release the memory allocated to the form. The form will be closed and the memory will be released when the corresponding 4GL window is closed by means of the CLOSE WINDOW statement.



OPEN WINDOW

The OPEN WINDOW statement is used to open a 4GL window.



Element	Description
Window Name	The name you want to assign to the new window
AT	An optional clause which specifies the position of the window
Window Size	The clause which specifies the size of the window
WITH FORM	The clause which specifies which form you want to open in the window
ATTRIBUTE Clause	Display attributes applied to the window

A 4GL window is an area on the screen where the associated menu, messages, errors, and screen forms are displayed. The OPEN WINDOW statement can include the information that specifies:

- the name of the window
- the position of the window
- the size of the window
- the display attributes of the window

The *window-name* must be unique among the other window names within the program.

The Window Stack

All the open 4GL windows are organized in a window stack. When a new window is opened by means of the OPEN WINDOW statement, the following actions are performed:

- The changes made in the current window are saved
- A new 4GL window is added to the window stack and is placed at the top of the stack above the current window
- The new 4GL window is made the current window

The window stack can be modified with the help of three statements: OPEN WINDOW, CURRENT WINDOW and CLOSE WINDOW.

The AT Clause

The AT clause contains two integers which are the co-ordinates of the top left corner of the window relative to the 4GL screen.





Element	Description
Lines	A 4GL expression which returns the number of a line in the 4GL screen
Columns	A 4GL expression which returns the number of column on the specified line in the 4GL screen

The location of a 4GL window is relative to the entire 4GL screen but it is not dependant on the position of other windows and does not influence their position.

The coordinates of the AT clause can be represented by 4GL expressions that return positive integers which must fit the following ranges:

- The first coordinate must be an integer from 1 to the maximum number of the lines of the 4GL screen. The window will begin at the line specified as the first coordinate
- The second coordinate must be an integer from 1 to the maximum number of the characters which the 4GL screen can contain in one line. The window will begin at the column specified as the second coordinate

The coordinates must be separated by comma. The example below opens a window on the second line of the 4GL screen at its second column (character).

```
OPEN WINDOW info_window AT 2,2 WITH 4 ROWS, 25 COLUMNS
```

The AT clause is obligatory regardless whether you open a window with form or without it.

Window Size

The window size can be specified either with the traditional WITH ROWS, COLUMNS clause or using the SIZE clause.



Element	Description
Lines	A 4GL expression which returns the number of a line in the 4GL screen
Columns	A 4GL expression which returns the number of column on the specified line in the 4GL screen

Lines and columns must be 4GL expressions that return positive integers which are not greater than the total size of the 4GL screen, which means they must obey the following rules:

- The 'lines' must be an integer from 1 to the maximum number of lines that can be displayed by the 4GL screen.
- The 'columns' must be an integer from 1 to the maximum number of characters that can be displayed on a single line of the 4GL screen.

The size of the 4GL screen in the GUI mode can be viewed and changed using the file containing environment variables, the variables that control the 4GL screen size are "LINES" and "COLUMNS". The size of the 4GL screen in character mode can be changed using the standard settings which are applied to the system command line environment.



The example below will open a 4GL window 10 lines high and 50 characters wide:

```
OPEN WINDOW my_window AT 2,10 WITH 10 ROWS, 50 COLUMNS
```

Remember, that there are several reserved lines that cannot be occupied by a screen form. You must set the size of a window taking into consideration these lines:

- The MESSAGE LINE is the second line of the window by default
- The FORM LINE is the third line of the window (it is the line where the top line of the form is located by default)
- The ERROR LINE is the last line of the 4GL window by default

If a 4GL window cannot accommodate all the necessary lines (if its size is smaller than the total sum of the form lines and all the reserved lines), a runtime error will occur. You can reduce the number of lines required by a window by specifying the FORM LINE as the first or second line of the window and change the positions of the other reserved lines as well by means of the ATTRIBUTE clause of the OPEN WINDOW statement.

The minimum number of lines required by a window to display a form is the sum total of the form lines and one line below the form for the error messages.

The WITH FORM Clause

The WITH FORM clause is an alternative to the Window Size clause which specifies the size of a window explicitly. The WITH FORM clause implicitly defines the number of rows and columns of the window depending on the dimensions of the form specified in the in this clause.

— **WITH FORM** → **Form File** →

Element	Description
Form File	A 4GL expression that returns the name of the form file and its relative path, if required

The form file which follows the WITH FORM keywords can be:

- A quoted character string containing the of the form file without extension and with path, if it is required.
- A variable, a record member or an array element of character data type that contains the quoted name of the form file without the extension and with optional relative path.
- Any 4GL expression that returns the quoted name of the form file without the extension and with optional relative path.

The height of the window opened with the WITH FORM clause is calculated according to these formulas:

- In the graphical display mode: $(form-line) + (form-length)+1$ line. The line added at the end is the comment line. If you do not include one line into the window for the comments, they will be hidden behind the last line of the form. The error line is added automatically, it is not included into the declared window size.
- In the character display mode $(form-line) + (form-length)$. The error line is added automatically, it is not included into the declared window size. The comment line coincides with it.

The form line is the line where the first line of the form is displayed; it is the third line of a window. The form length is the number of lines in the SCREEN section of the form specification file.



Unless other line is specified as the form line in the OPTIONS statement or in the ATTRIBUTE clause of the corresponding window, the default value of this sum in the character display mode is $3 + (\text{form-length})$.

The example below opens a window *my_window* at the third line and sixth column of the 4GL screen with form *my_form*. The height of the *my_form* is 15 lines, thus the total height of the window with the default FORM LINE position will be $3 + 15 = 18$ lines:

```
OPEN WINDOW my_window AT 3,6 WITH FORM "my_form"
```

Opening a window with a form is convenient, if a window should display only one form throughout the program. You do not need the OPEN FORM, DISPLAY FORM, or CLOSE FORM statements to operate the form, if you open it by means of the OPEN WINDOW statement. The OPEN WINDOW statement opens and displays a form at the same time. The CLOSE WINDOW statement closes the form together with the window.

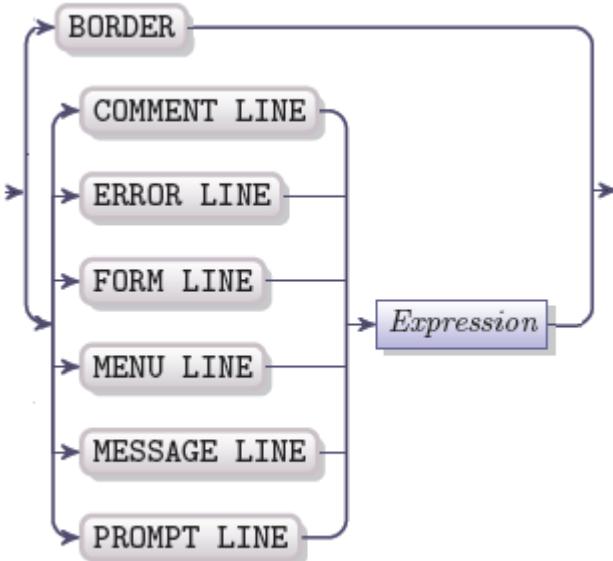
The OPEN WINDOW ... WITH FORM statement cannot be used for:

- Displaying more than one form in one 4GL window
- Displaying a 4GL window with larger dimensions than the dimensions of the form specified in the statement

To perform the actions listed above use the OPEN WINDOW ... WITH ROWS, COLUMNS statement and use the OPEN FORM, DISPLAY FORM, or CLOSE FORM statements to operate a form or forms.

The ATTRIBUTE Clause

The ATTRIBUTE clause of the OPEN WINDOW statement has the structure of the common [ATTRIBUTE clause](#). The ATTRIBUTE clause of the OPEN WINDOW statement can include the following special attributes which are not mentioned in the common ATTRIBUTE clause section:



Element	Description
---------	-------------



Expression

A 4GL expression that returns a positive integer or a keyword

The expression must return the number of line within the 4GL window on which the corresponding information must be displayed. This number cannot be smaller than 1 and greater than the total number of lines in the window. If you specify the COMMENT LINE attribute, it can have an additional value – OFF – which hides the comment line.

Here is the list of all the attributes which can be used in the OPEN WINDOW statement:

Colour attributes	Line attributes	Other attributes
WHITE	PROMPT LINE	BORDER
YELLOW	FORM LINE	REVERSE
RED	MESSAGE LINE	NORMAL
MAGENTA	MENU LINE	BOLD
BLUE	COMMENT LINE	DIM
GREEN		
CYAN		
BLACK		



Note: The position of an error line cannot be changed by means of the OPEN WINDOW statement. No error will occur if you try to do it, but it will have no effect. It can be modified only by means of the OPTIONS statement.

There are the default settings for these attributes, which are applied when the attribute is not specified:

- A colour attribute – the default colour of your terminal
- The REVERSE attribute – no reverse video
- The FORM LINE – third line of the window
- The MENU LINE – the first line of the window is reserved for menu options, the second line is reserved for options descriptions
- The MESSAGE LINE – second line of the window
- The PROMPT LINE – first line of the window
- The COMMENT line – the last line of the window in the character display mode and the last but one line in the graphical display mode
- The BORDER attribute – no border

The colour attribute or REVERSE attribute specified in the ATTRIBUTE clause is used as the default attribute for displaying all the text within the window except for the menu. To override these settings use the ATTRIBUTE clauses of the statements used to display information in this window. CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY FORM, INPUT, or INPUT ARRAY statements can have the ATTRIBUTE clauses which will override the OPEN WINDOW attributes.

The BORDER Attribute

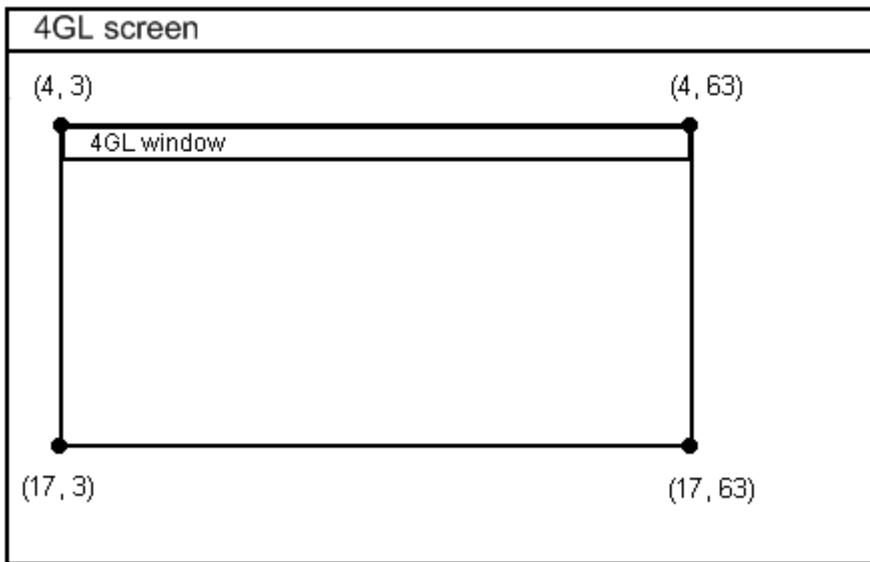
The BORDER attribute specifies that a 4GL window should be displayed with the outside border. The border adds two more lines to the window dimensions (at the top and at the bottom) and two more characters (at the left and at the right side). Take the additional dimensions into account when you specify the position of the window in the AT clause. The AT clause positions the inner corner of a window ignoring the border.

```
OPEN WINDOW my_window3 AT 5, 4 WITH 12 ROWS, 59 COLUMNS
```



ATTRIBUTE (BORDER)

The window will be positioned regarded to the 4GL screen as follows:



The coordinates of the top left corner are (4, 3), though the window has been declared with coordinates (5, 4) – one character has been added before the first declared character of the window and one line has been added before the first declared line of the window due to the BORDER attribute. The window takes up 14 lines and 61 characters, though the 4GL window itself is 12 lines and 59 characters.

The LINE Attribute

The reserved lines can be positioned either in the OPTIONS statement or in the ATTRIBUTE clause of the OPEN WINDOW statement. The LINE attribute contains of the keywords that specify the reserved line (PROMPT LINE, MENU LINE, MESSAGE LINE, FORM LINE) followed by:

- A literal positive integer
- A variable that returns a literal positive integer
- FIRST keyword
- FIRST + *integer* (where *integer* is a literal positive integer or a variable that returns such integer)
- LAST keyword
- LAST - *integer* (where *integer* is a literal positive integer or a variable that returns such integer)

```
OPEN WINDOW my_window4 AT 2, 2 WITH FORM "my_form"  
ATTRIBUTE (PROMPT LINE LAST-2, MESSAGE LINE FIRST+4, FORM LINE 6)
```

The above example positions the Prompt line on the line which is two lines higher than the last line of the window, the message line at the fifth line and the form line at the sixth line of the window.

If the LINE attribute for a reserved line is not set, the default line is used for its displaying, unless the other settings have been specified in the OPTIONS statement. By default the reserved lines are positioned as follows:



- The first line – the PROMPT LINE and the first MENU LINE where the menu options are positioned
- The second line – the MESSAGE LINE and the second MENU LINE where the descriptions are located
- The third line – the FORM LINE which is the position of the first line of the displayed screen form
- The last but one line - the COMMENT line, if the program is run in the graphical display mode
- The last line – the ERROR LINE and the COMMENT line, if the program is run in the character display mode

You must take into consideration the following restrictions when you specify values for the reserved lines:

- You cannot specify the LAST line as the MENU LINE, because a menu requires two lines to be displayed: the first displays the menu options and the second displays their descriptions
- Do not specify the LAST or LAST+*integer* for the FORM LINE, otherwise 4GL may not be able to display the form due to insufficient space.

If 4GL cannot display the value of one or more of the reserved lines because the window is not large enough, it moves the value either to the FIRST line of the window or to the LAST, depending on which is appropriate.

All the values of the reserved lines can be displayed only on the specified lines. They cannot take up two or more lines, if the value is longer than the window width. The value is truncated from the right, if it is longer than the corresponding line.

The INPUT statement clears the error line, so do not position the MESSAGE or PROMPT lines at the LAST line of a window, unless you have relocated the ERROR LINE with the help of the OPTIONS statement.

Redirecting Errors, Messages and Comments

You may want to display an error, a message or a comment not inside a window but outside its usable area so that it didn't overlap with the window or form contents. You can use the following keywords instead of specifying the line of the window in the MESSAGE LINE, ERROR LINE, and COMMENT LINE attributes:

- MESSAGEBOX - displays the message, error or comment to a separate message box that pops up when MESSAGE or ERROR statements are executes or when the cursor enters the field with a comment correspondingly.
- STATUSBAR - displays the message, error or comment to the status bar of the current window when MESSAGE or ERROR statements are executes or when the cursor enters the field with a comment correspondingly.

Here is an example of the redirection used:

```
OPEN WINDOW w1 AT 2,2 WITH FORM "myform"  
ATTRIBUTE (BORDER, ERROR LINE STATUSBAR, MESSAGE LINE MESSAGEBOX)
```

Hiding the COMMENT Line

By default the comment line is the last line of the 4GL window and the last but one line of the 4GL screen. If you run the program with the help of a graphical client, the comment of a form field will be shown in a pop-up message when the mouse cursor points at it, in addition to the text in the comment line.

The position of the comment line can be changed by means of the ATTRIBUTE clause of the OPEN WINDOW statement and by means of the OPTIONS statement. You can prevent the comments from being displayed by specifying the OFF value for the COMMENT LINE option of the ATTRIBUTE clause.

```
OPEN WINDOW my_window4 AT 2, 2 WITH FORM "my_form"  
ATTRIBUTE (COMMENT LINE OFF)
```



The comments will not be displayed at the comment line and they will not popup when the cursor points at the field with the COMMENT attribute, if the comment line is hidden.

Manipulating Windows at the Runtime Using Graphical Clients

If you run the program with the help of a thin graphical client, the border of the window will take up more than one line around the window due to the tile bar of the window which corresponds to the standard view of a window in the Windows OS and where the name of the window is written.

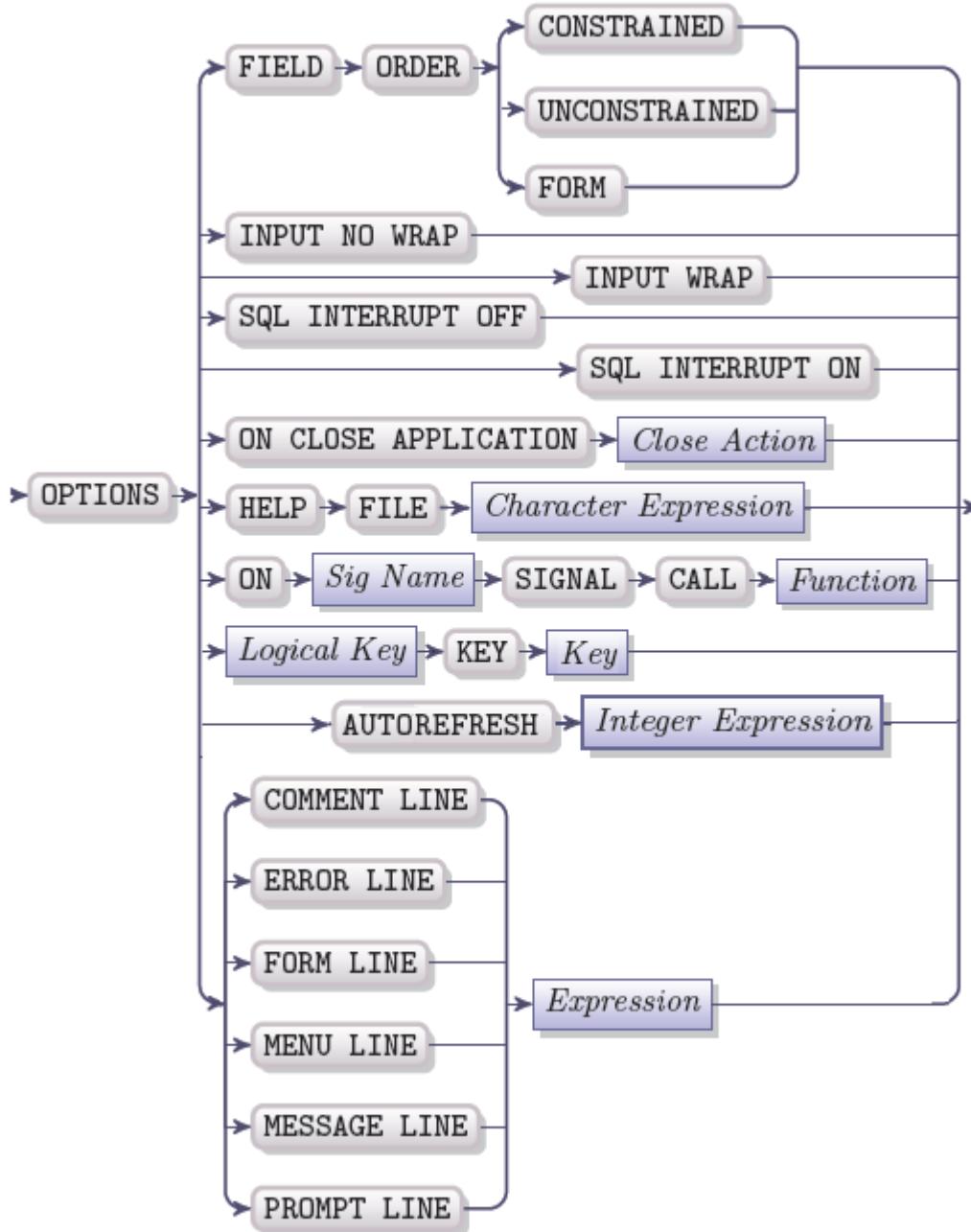
The user can manipulate the window in the same way as it is done in the Windows OS:

- The user can move the window around the screen in a habitual way by clicking and holding the left mouse button on the window tile bar.
- The size of a window can be changed dynamically at the runtime by clicking and holding the left mouse button on the borders of the window and then dragging the mouse cursor.
- The window can be closed, minimized and maximized with the help of the corresponding buttons in the upper right corner of the window.



OPTIONS

The OPTIONS statement is used to modify the default display attributes and default features of the screen interaction statements (CONSTRUCT, DISPLAY, DISPLAY ARRAY, DISPLAY, FORM, ERROR, INPUT, INPUT ARRAY, MESSAGE, OPEN FORM, OPEN, WINDOW, PROMPT, REPORT, RUN, and START REPORT).



Element	Description
Close Action	An event handler used when the application is closed
Character Expression	An expression that returns the name of the help file



Sig Name	The name of a termination signal
Function	The name of a function called by the ON ... SIGNAL clause
Logical Key	A logical key used in 4GL (ACCEPT, QUIT, etc)
Key	The name of a key or a key combination you want to assign to a logical key
Integer Expression	The identification of the refresh mode. Can be one of the following: 1,2, or 3.
Expression	An integer expression returning a positive integer or an expression containing keywords specifying the destination

One program module can include several OPTIONS statements; the most recent OPTIONS statement overrides the settings of the previous ones, if they specify the same features.

If the OPTIONS statement is omitted or if a clause is not specified, the clauses of the OPTIONS statement have the following default effects:

Clause	Default effect
COMMENT LINE	Used to specify the position of the comment text supplied by the COMMENT attribute of a form field. The default position of the comment line is the last line of the 4GL window in the character mode
ERROR LINE	Used to specify the position of the message produced by the ERROR statement. By default it is the LAST line of all the 4GL windows and of the 4GL screen
FORM LINE	Used to specify the position of the first line of a screen form displayed in a window. By default it is the FIRST+2 (third) line of a 4GL window
MENU LINE	Used to specify the position of the menu. By default it is the FIRST line of a 4GL window. The line below the first MENU line which displays menu options is occupied by the menu descriptions. So MENU statement actually occupies two lines, though only the position of the first one can be defined.
MESSAGE LINE	Used to specify the position of the message produced by the MESSAGE statement. By default it is the FIRST+1 (second) line of a 4GL window
PROMPT LINE	Used to specify the position of the message produced by the PROMPT statement. By default it is the FIRST line of a 4GL widow
ACCEPT KEY	Used to specify a key that terminates the CONSTRUCT, INPUT, INPUT ARRAY, or DISPLAY ARRAY statements. By default it is ESCAPE
DELETE KEY	Used to specify a key used to delete a screen record when an INPUT ARRAY statement is in effect. By default it is F2
INSERT KEY	Used to specify a key used to insert a screen record when an INPUT ARRAY statement is in effect. By default it is F1
NEXT KEY	Used to specify a key used to scroll to the next page of records in an INPUT ARRAY or DISPLAY ARRAY statement. By default it is F3.
PREVIOUS KEY	Used to specify a key used to scroll to the previous page of records in an INPUT ARRAY or DISPLAY ARRAY statement. By default it is F4
HELP KEY	Used to specify a key used to invoke and display help messages. By default it is CONTROL-W
HELP FILE	This clause specifies the help file that contains help messages used in the program. It does not have a default value.
ON CLOSE	Used to specify an event handler activated when an application is



APPLICATION ON ... SIGNAL	closed. The default event handler is STOP Used to specify an action 4GL undertakes when the specified signal is passed by the application
DISPLAY ATTRIBUTE	Used to specify a set of attributes applied to the DISPLAY and DISPLAY ARRAY statements. They are overridden by the attributes specified directly in the ATTRIBUTE clauses of these statements or by the attributes in the form specification file
INPUT ATTRIBUTE	Used to specify a set of attributes applied to the INPUT and INPUT ARRAY statements. They are overridden by the attributes specified directly in the ATTRIBUTE clauses of these statements or by the attributes in the form specification file. If the user presses RETURN key at the last field of the form, the form is not deactivated
INPUT WRAP	Specifies that the screen cursor will wrap from the last field of the screen array to the first one during the INPUT, INPUT ARRAY, and CONSTRUCT statements until the Accept key is pressed. If the user presses RETURN key at the last field of the form, the form is not deactivated. This is not the default value.
INPUT NO WRAP	Specifies that the screen cursor does not wrap to the first field of the form from the last one and the form is deactivated when the RETURN key is pressed. This is the default value.
FIELD ORDER CONSTRAINED	When this clause is activated the UP ARROW and DOWN ARROW keys move the cursor from the current field to the previous field and next field (respectively) during the INPUT or CONSTRUCT statements
FIELD ORDER UNCONSTRAINED	When this clause is activated the UP ARROW and DOWN ARROW keys move the cursor to the field above and below the current field (respectively) during the INPUT or CONSTRUCT statements
FIELD ORDER FORM	The clause makes the cursor move from field to field according to the TABINDEX form fields attribute. If the tab index of a field is set to zero, the field is excluded from the tabbing, but you can still focus on it using the mouse.
SQL INTERRUPT ON	Specifies that the user is able to interrupt SQL statements
SQL INTERRUPT OFF	Specifies that the user is not able to interrupt SQL statements
AUTO REFRESH	Specifies that the output of the DISPLAY statement (DISPLAY TO, DISPLAY AT, DISPLAY FORM, etc.) will be displayed immediately after the statement is executed and will not wait for the next user interaction statement.
MANUAL REFRESH	Specifies that the output of the DISPLAY statement (DISPLAY TO, DISPLAY AT, DISPLAY FORM, etc.) will be displayed not directly after the statement is executed, but after the next user interaction statement suspends the program and waits for the user input. This is the default option.

The Reserved Lines

All the reserved lines can be repositioned by the OPTIONS statement by means of the LINE clause. The LINE clause contains of the keywords that specify the reserved line (PROMPT LINE, MENU LINE, MESSAGE LINE, FORM LINE, ERROR LINE, COMMENT LINE) followed by:

- A literal positive integer
- A variable that returns a literal positive integer
- FIRST keyword
- FIRST + *integer* (where *integer* is a literal positive integer or a variable that returns such integer)



- LAST keyword
- LAST - *integer* (where integer is a literal positive integer or a variable that returns such integer)

You must take into consideration the following restrictions when you specify values for the reserved lines:

- You cannot specify the LAST line as the MENU LINE, because a menu requires two lines to be displayed, the first displays the menu options and the second displays their descriptions
- Do not specify the LAST or LAST + *integer* for the FORM LINE, otherwise 4GL may not be able to display form due to insufficient space.

If 4GL cannot display the value of one or more of the reserved lines because the window is not large enough, it moves the value either to the FIRST line of the window or to the LAST, depending on which is appropriate.

All the values of the reserved lines can be displayed only on the specified lines. They cannot take up two or more lines, if the value is longer than the window width. The value is truncated from the right, if it is longer than the corresponding line.

The INPUT statement clears the error line, so do not position the MESSAGE or PROMPT lines at the LAST line of a window, unless you have relocated the ERROR LINE with the help of the OPTIONS statement.

The positions of the reserved lines specified in the OPTIONS statement remain valid until 4GL encounters another OPTIONS statement, which redefines the positions of lines or until it encounters the OPEN WINDOW statement which changes these settings. After the window opened by the OPEN WINDOW statement is closed the effect of the most recent OPTIONS statement is restored.

Redirecting Errors, Messages and Comments

You may want to display an error, a message or a comment not inside a window but outside its usable area so that it didn't overlap with the window or form contents. You can use the following keywords instead of specifying the line of the window in the OPTIONS MESSAGE LINE, OPTIONS ERROR LINE, and OPTIONS COMMENT LINE statements:

- MESSAGEBOX - displays the message, error or comment to a separate message box that pops up when MESSAGE or ERROR statements are executed or when the cursor enters the field with a comment correspondingly.
- STATUSBAR - displays the message, error or comment to the left part of the status bar line of the current window when MESSAGE or ERROR statements are executed or when the cursor enters the field with a comment correspondingly.
- ALTSTATUSBAR - displays the message, error or comment to the right part of the status bar line of the current window when MESSAGE or ERROR statements are executed or when the cursor enters the field with a comment correspondingly.

Here is an example of the redirection used:

```
OPTIONS  
    ERROR LINE MESSAGEBOX,  
    MESSAGE LINE STARUSBAR  
    ...  
    MESSAGE "I'm displayed to the statusbar."  
    ...
```



ERROR "I'm displayed to the message box."

The Cursor Movement Settings

The order in which the screen cursor is moved from one field of a form to another during the CONSTRUCT, INPUT, or INPUT ARRAY statement depends on the *field-list* used in these statements. These statements are terminated when the RETURN key is pressed at the last field of the form.

The INPUT WRAP clause of the OPTIONS statement changes this default behaviour. It makes the cursor return to the first field of the form when the RETURN key is pressed, if the cursor is at the last field of the form. Use the INPUT NO WRAP clause to restore the default behaviour of the cursor.

The clause FIELD ORDER UNCONSTRAINT changes the reaction of the cursor to the arrow keys as described at the previous page, it is not the default behaviour. The FIELD ORDER CONSTRAINT can be used to restore the default behaviour of the cursor. The FIELD ORDER FORM option makes the cursor move according to the form fields Tabindex attribute values. Note, that the cursor cannot be reached by next/previous keys if its Tabindex is set to 0; however, the user can use mouse to focus on it.

The ATTRIBUTE Clause of the OPTIONS Statement

There are two clauses that can be regarded as the ATTRIBUTE clause of the OPTIONS statement; they are DISPLAY ATTRIBUTE and INPUT ATTRIBUTE. They are used to specify attributes for the DISPLAY and DISPLAY ARRAY statements and for the INPUT and INPUT ARRAY statements respectively. The initial keywords differ but the attributes that can be used in them are the same, so they both will be regarded as the ATTRIBUTE clause.

The ATTRIBUTE clause of the OPTIONS statement can specify the following:

- The foreground attributes of a 4GL window
- Whether 4GL should use the input attributes of the current form or 4GL window
- Whether 4GL should use the display attributes of the current form or 4GL window

```
OPTIONS DISPLAY ATTRIBUTE (RED, REVERSE)
```

The attributes specified in the ATTRIBUTE clauses of the OPEN WINDOW, CONSTRUCT, INPUT, DISPLAY, and DISPLAY ARRAY statements redefine the attributes only in their scope of reference. E.g. when a window is opened, the ATTRIBUTE clause of the OPEN WINDOW statement overrides the attributes of the most recent OPTIONS clause. However, when the window is closed, the attributes previously set by the OPTIONS statement are restored. The attributes of an OPTION statement placed after the OPEN WINDOW statement will override the settings of the OPEN WINDOW statement.

If the FORM keyword is used in the ATTRIBUTE clause of the OPTIONS statement, it means that the display attributes of the current form are used:

```
OPTIONS INPUT ATTRIBUTE (FORM)
```

The WINDOW keyword can be used in these clauses to specify that the display attributes of the current 4GL window should be used. The FORM and WINDOW keywords cannot be used with any other keywords within one ATTRIBUTE clause.



OPTIONS DISPLAY ATTRIBUTE (WINDOW)

The HELP FILE Clause

This clause is used to specify the name of the help file (either in the quoted string or as a variable which returns such quoted string). The filename can include a pathname if necessary and it must include the extention of the compiled help file. Pay attention to the fact that the extention of the compiled help file differs from the extention of the initial help file. E.g. the extention of the help file you create is .msg, but the compiled file has the extention .erm.

```
HELP FILE "/source/msg_files/my_help_file.erm"
```

Assignment of the Keys

The OPTIONS statement is also used to specify the physical keys for some standard 4GL actions. To change the default keys for the standard actions, you must specify the logical key for such action (e.g. ACCEPT KEY).

There are the following logical keys which can be used in the OPTIONS statement:

- ACCEPT KEY
- DELETE KEY
- INSERT KEY
- NEXT KEY
- PREVIOUS KEY
- HELP KEY

The logical key must be followed by the name of the key, they are:

DOWN	NEXT/NEXTPAGE
ESC/ESCAPE	PREVIOUS/PREVPAGE
INTERRUPT	RETURN/ENTER
LEFT	TAB
RIGHT	UP
F1 – F256	CONTROL- <i>char</i> (except A, D, H, I, J, K, L, M, R, or X)

```
OPTIONS  
    ACCEPT KEY ENTER,  
    HELP KEY F1
```

Some keys require special consideration when used in the OPTIONS statement:

Key	Usage Features
ESC/ESCAPE	If you assign the ESC/ESCAPE to a 4GL action other than ACCEPT, you must specify other key for the ACCEPT action, because the ESC/ESCAPE key is the default key for the ACCEPT action
INTERRUPT	The DEFER INTERRUPT statement must be executed in order that this key could be used in the ON KEY block. On pressing the Interrupt key the corresponding ON KEY block is executed, int_flag is set to non-zero



QUIT	The DEFER QUIT statement must be executed in order that this key could be used in the ON KEY block. On pressing the Quit key the corresponding ON KEY block is executed and <code>quit_flag</code> is set to non-zero
CTRL-char (A, D, H, K, L, R, X)	4GL reserves these keys for field editing
CTRL-char (I, J, M)	These key combinations by default mean TAB, NEWLINE and RETURN. If they are used in the OPTIONS statement, they cannot perform their default functions. Thus, if you use these keys in the OPTIONS statement to assign other functions to them, the scope of reference of such OPTIONS statement should not be large.

Some other keys may have special meanings in your operating system and you may not be able to use them in the OPTIONS statement. E.g. keys CONTROL-C, CONTROL-Q, and CONTROL-S are often used to specify the Interrupt, XON, and XOFF signals.

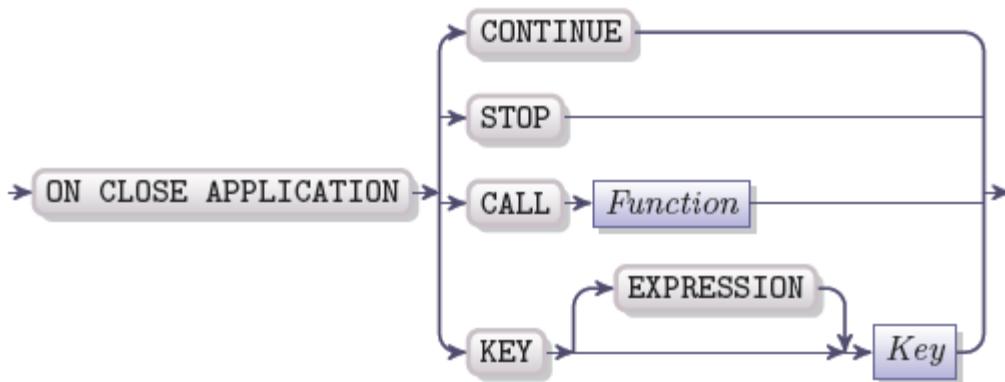
You can disable a 4GL action by assigning it to a sequence of keys that are recognized not as the action key but as the editing keys. E.g. If you assign an action to the CTRL-char (where char is A, D, H, K, L, R, or X), the action will never be executed and the keys will be used as the editing keys. The example below disables the INSERT key:

```
OPTIONS INSERT KEY CONTROL-D
```

The user is not able to insert screen records during the INPUT ARRAY statement, if this OPTIONS statement precedes the INPUT ARRAY statement.

The ON CLOSE APPLICATION and ON ... SIGNAL Clauses

The ON CLOSE APPLICATION clause is used to specify the action which 4GL should undertake when the user presses the close button at the top right corner of the application window.



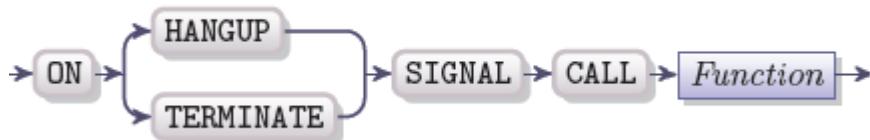
Element	Description
Function	The name of a function called by the ON ... SIGNAL clause
Key	The name of a key or button you want to activate when the application is being closed



You can use the KEY event handler to activate a key or a form button to which some action is assigned.

OPTION ON APPLICATION CLOSE CONTINUE

The ON... SIGNAL clause has similar function. It specifies one of the signals that can be passed by the application and defines the event handler used to deal with the signal:



Element	Description
Function	A function name

This clause is applied when 4GL receives one of the following signals:

- TERMINATE - an application has been terminated, works only with terminals and terminal emulations
- HANGUP - the connection to the client or terminal has been closed

The AUTOREFRESH

The productivity and performance of a program can be improved by managing the refreshment modes. The first refreshment mode allows you to reach maximum performance, since the program does not need to display anything until it waits for the user input and thus the calculations can go faster.

Lycia offers three refreshment modes:

- The output of all the DISPLAY statements (DISPLAY, DISPLAY TO, DISPLAY AT, DISPLAY FORM) and the visual indication of adding or removing a row to an array will be visible only when the program executes a user interaction statement located further in the code and waits for the user input. The user indication statements can be INPUT, INPUT ARRAY, DISPLAY ARRAY, MENU, PROMPT and others. To set this refreshment mode, assign value 1 to AUTOREFRESH:

OPTIONS AUTOREFRESH 1

- The output of the DISPLAY statements and alike will be visible either during the user interaction statements, or if the SLEEP statement was executed. This is the default refreshment mode. To invoke this mode, set the value to 2.

OPTIONS AUTOREFRESH 2

- The output of all DISPLAY statements will be visible immediately after they are executed and will not wait for user interaction statements to be in effect. This mode is the slowest of the three and gives the worse performance, but at the same time it gives your program an improved feedback mechanism. To enable this options set the value to 3.

OPTIONS AUTOREFRESH 3



If the first refreshment mode is on, the long SQL queries or other time consuming processes will prevent anything from displaying as long as they are running. If you want to see the output of the display statements while no user interaction statement is in effect, call the ui.interface.refresh() method. It will show the output of all displays executed between the previous user interaction statement and the place this method was called at.

Thus, if you want to keep track of the long database interactions, for example, and display the progress, you should either use the third or second refreshment mode, or use the refresh() method after displaying.

The SQL INTERRUPT ON/OFF Clauses

The SQL INTERRUPT clause is used to allow or forbid the user to interrupt the SQL statements in the same way as 4GL statements. The default value of this option is OFF; this means that SQL statements cannot be interrupted by pressing Interrupt key. If this option is not activated, 4GL waits until SQL server finishes the execution of the SQL statement even if the Interrupt key has been pressed. Only after the server has completed its work, the following actions are performed:

- The int_flag is set to TRUE and the execution is continued, if the DEFER INTERRUPT statement is in effect
- the program is terminated, if the DEFER INTERRUPT statement is not in effect

In order to allow the user to interrupt SQL statements specify the SQL INTERRUPT ON clause in the OPTIONS statement. The DEFER INTERRUPT statement must be in effect. When these conditions are met the following actions take place:

1. The SQL statement is terminated. The following statements can be terminated in such way:

- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DELETE
- FETCH – including the FOREACH loop that contains an implicit FETCH statement
- INSERT – includes INSERT that takes place during the LOAD statement
- OPEN – is interrupted if the SELECT statement stores data in a temporary table
- SELECT - includes SELECT that takes place during the UNLOAD statement
- UPDATE

If the SQL statement is located within a transaction, the database server must handle the interrupted transaction.

2. The int_flag is set to TRUE
3. The status built-in variable is set to -213 (it is an error code)
4. The statement that follows the interrupted SQL statement is executed, if you have specified WHENEVER ERROR CONTINUE statement in your program. Otherwise the program is terminated.

SQL server completes the execution of the SQL statements that are not in the list of statements above, if the Interrupt key is pressed. After that the int_flag variable is set and the execution of the program is continued from the first statement following such SQL statement, if you have specified the WHENEVER ERROR CONTINUE statement in your program.

The same actions are performed if the Quit key is pressed and if the DEFER QUIT statement is in effect. The only difference in the 4GL behaviour is that the quit_flag is set instead of the int_flag.



The default effect of the SQL INTERRUPT clause can be cancelled by the OPTIONS statement that is located later in the program and that contains the SQL INTERRUPT OFF clause.

The Transactions Interruption

If an SQL statement is interrupted, it has certain consequences for database transactions. If the database does not support transaction logging, the effect of the SQL INTERRUPT ON clause is very limited. The reaction of the database to this clause depends on whether it is ANSI-compliant and on the type of the transaction:

Database Type	Transaction Type	Description
non-ANSI-compliant	explicit	Begins with the BEGIN WORK statement and ends with the COMMIT WORK statement or ROLLBACK WORK statement
	singleton	An SQL statement that is not placed within an explicit transaction forms a transaction of its own, the transaction ends when the statement completes
ANSI-compliant	implicit	Transactions are always in effect

Explicit Transaction Interruption (how does it work with other databases)

An explicit transaction is a set of statements between the BEGIN WORK statement and the COMMIT WORK statement (or sometimes the ROLLBACK WORK statement). When the user presses the Interrupt key or Quit key the following actions are performed:

- IBM Informix Dynamic Server – all interruptible SQL statements are interrupted and undone automatically
- IBM Informix SE - all interruptible SQL statements (except for ALTER INDEX and CREATE INDEX) are interrupted but are not undone automatically, the transaction will still be in progress

Interruption of the Singleton Transactions

A singleton transaction can occur only in a non-ANSI-compliant database. Each SQL statement that is not included into an explicit transaction is executed in a singleton transaction. In a database that uses transaction logging the BEGIN WORK statement is necessary to start a transaction.

If an SQL statement that can be interrupted is located in a singleton transaction and the user presses the Interrupt key, the changes made by the current SQL statement are rolled back and the control is returned to the 4GL. No transaction will be in progress.

Interruption of the Implicit Transactions

The BEGIN WORK statement is not required in the implicit transactions. The new transaction immediately follows the COMMIT WORK or ROLLBACK WORK statements. In an ANSI-compliant database an SQL statement cannot be executed outside a transaction. When an implicit transaction is interrupted, the ROLLBACK WORK is not used automatically. The transaction will still be in progress.

Managing Interruption of the Transactions

If no rollback is performed when a transaction is interrupted, the database can be left in an unknown state. There are two ways to check for an interrupted SQL statement:



- Test the int_flag built-in variable (if the DEFER INTERRUPT statement is in effect). Its value is TRUE, if the user has pressed the Interrupt key during the SQL statement
- Test the status built-in variable (if the WHENEVER ERROR CONTINUE statement is in effect). Its value is -213 if the user has pressed the Interrupt key during the SQL statement and the statement has been interrupted

If the database is in the unknown state after the interruption of a transaction, specify the ROLLBACK WORK explicitly. It will reverse the current transaction. If you use the COMMIT WORK statement, all changes and modifications made will be saved in the database. However, you should avoid using this statement if the database is in the unknown state. The BEGIN WORK statement should be used to start a new transaction.

The ROLLBACK WORK statement reverses the current transaction and automatically opens a new one in an ANSI-compliant database.

Default Screen Display Modes

There are two screen display modes:

- IN LINE MODE
- IN FORM MODE

These modes can be specified explicitly in the following statements: OPTIONS, RUN, REPORT, and START REPORT. Different screen display modes can be specified for the RUN statement and for the REPORT statement that sends the output to the pipe by means of the OPTIONS statement.

If the IN LINE MODE is activated, the terminal is in the same state as when the program began. The interrupts are enabled and the terminal input is in the cooked mode. The input does not become available until a NEWLINE character is typed.

If the IN FORM MODE is activated the terminal input is performed in raw mode, each character is available for input the moment it is typed.

The IN LINE MODE is the default screen display mode. However, many 4GL statements take it into IN FORM MODE (these are DISPLAY AT, OPEN WINDOW, DISPLAY FORM statements, OPTIONS statement that specifies keys, etc.). That is why many of the 4GL programs are in the IN FORM MODE most of the time.

The IN FORM MODE is the default mode for PIPE. The default screen display mode for RUN is IN LINE MODE. If you try to specify RUN IN FORM MODE in the OPTIONS statement, the program will not enter the formatted mode. However, if it is already in formatted mode, it will remain formatted. The RUN IN LINE MODE switches the program to the line mode, if it has been in formatted mode.



OUTPUT TO REPORT

The OUTPUT TO RECORD statement is used to pass an input record (a set of values) to a RECORD statement. By means of this statement, 4GL is instructed to format these values and process them as the next input record of the RECORD statement.



Element	Description
Report Name	The name of the currently processed report
Input Record	The comma separated arguments passed to the report

An input record is a set of values returned by expressions specified within the parentheses of the OUTPUT TO REPORT statement. The values are ordered in accordance with the order of these expressions. An input record can be the same as a program record or a database row but it does not have to correspond to any of them.

The members of the input record within the parentheses of the OUTPUT TO RECORD statement must match the formal arguments of the REPORT statement in number, order and they must be of compatible data types.

You must remember that the arguments of large data types (BYTE or TEXT) are passed by reference, not by values like the arguments of other data types. The values of TEXT data types can be displayed with the help of the PRINT statement with the WORDWRAP operator. The values of BYTE data types cannot be displayed in a report.

The OUTPUT TO REPORT statement is typically used in a loop (FOR, WHILE, or FOREACH). In that case the input records are placed into the report one at a time. The code sample below performs the input of the records to a report in a loop:

```
START REPORT order_list
...
FOREACH ord_cursor INTO order.ord_id, order.customer
    OUTPUT TO REPORT order_list (order.ord_id, order.customer,
        TODAY, manager12)
END FOREACH
```

Each input record produced by this example will consist of four members:

- order.ord_id, order.customer – these are the columns of the database table
- TODAY – this is the operator that returns the current date



- manager12 – this is the variable of character data type

If the OUTPUT TO REPORT statement is not executed within the REPORT program block, the control clauses of the report definition are also not executed, even if you include the START REPORT and FINISH REPORT statements into your REPORT program block.



PAUSE

The PAUSE statement is used to suspend the output from the report. It has no effect unless the output is sent to the screen and only if the 4GL application is run under Linux/Unix OS. On Windows, the PAUSE statement produces no effect even if the output is sent to the screen.



Element	Description
<i>String Expression</i>	A character expression that returns the message displayed by the PAUSE statement

The PAUSE statement on Linux/UNIX

If the PAUSE statement is included, the report output remains on the screen until the RETURN key is pressed.

The string expression following the PAUSE statement can be a quoted character string or any 4GL expression that returns such string.

If the TO clause is present in the START REPORT statement and is not followed by the SCREEN keyword, the PAUSE statement has no effect. Likewise, if the OUTPUT section contains the REPORT TO keywords and they are not followed by the SCREEN keyword, the PAUSE statement does not influence the output.

The PAUSE statement is typically included into the PAGE HEADER or PAGE TRAILER clause. In the example below, it is included into the PAGE HEADER clause, thus the new page of the report will be displayed only after the user presses the RETURN key.

```
PAGE TRAILER  
PAUSE "Press RETURN to view the next page"
```

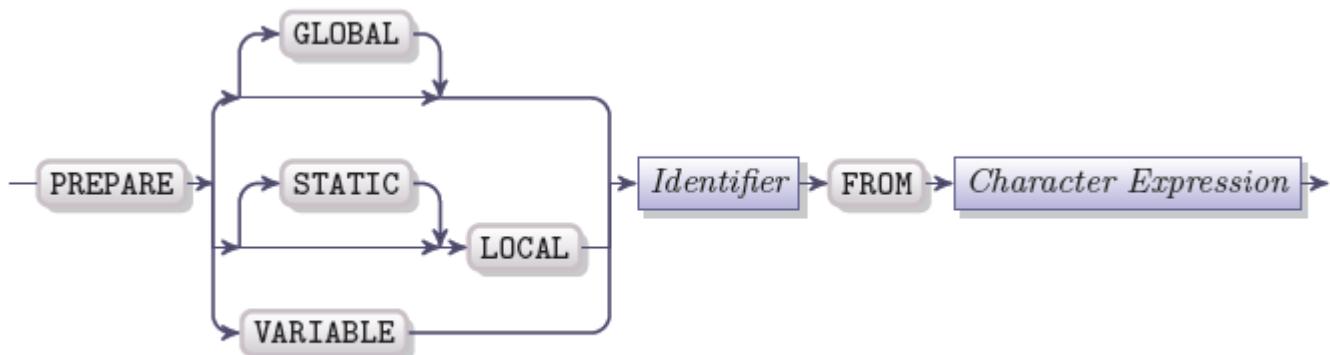
Output to the Screen on Windows

Querix has designed the Report Viewer, it is a tool which is a part of the graphical clients. This tool is used for displaying the output sent to the screen. It contains the complete report, the pages can be viewed by means of the left and right arrow buttons on the toolbar of the Report Viewer. Thus you can view any page of the report at any time. In the character mode the report is sent to the Notepad an instance of which is opened automatically.



PREPARE

The PREPARE statement is used to prepare an execution plan according to which SQL statements will be executed in a 4GL program.



Element	Description
Identifier	SQL Statement identifier, this must be unique among the declared cursors and other prepared statements
Character Expression	A character expression which returns the text of one or more SQL statements to be prepared

The PREPARE statement brings together the text and creates an SQL statement out of it at the runtime. The dynamic SQL is assembled in the following way:

- The text of the SQL statement is specified in the PREPARE statement as a character expression. The text can contain the question marks (?) which represent the user-supplied values which should be specified at runtime. This text can be either in the form of
 - a quoted character string
 - a variable of character type
- The prepared SQL statement can be executed by means of the EXECUTE or OPEN statement, the input values can be supplied by the USING clause. A prepared statement can be executed unlimited number of times
- After you have used the prepared statement and are sure you will not need it again, you can use the FREE statement to deallocate the resources used for the PREPARE statement.

See the sections of this chapter which touch upon the EXECUTE, OPEN, FREE and SQL statements for more information.



	<p>Note: A 4GL variable cannot be referenced within the PREPARE statement. To do so, use the SQL ... END SQL statement instead.</p>
---	--

The number of prepared objects in a program is not limited by 4GL. It is only limited by the free memory on your system. A prepared object can include cursor declarations (that include SELECT, EXECUTE PROCEDURE, or INSERT statements) and statements identifiers.

The Statement Identifier

At runtime, the text of the PREPARE statement is passed to the database server. In case the text is a valid SQL statement and if it does not contain syntax errors, it is transformed into an internal form. The transformed statement is saved in a data structure that the PREPARE statement allocates, so that it could be used repeatedly later. The saved structure has the name which is the identifier of the PREPARE statement. This identifier can be referenced by other statements to invoke the prepared structure.

The default scope of reference of the PREPARE statement is the program module. In some database servers, the identifier of the prepared statement cannot be longer than 9 symbols. The prepared statement is referenced by its identifier; the identifier cannot be referenced from another program module.

The scope of reference of the PREPARE statement can be also widened, if the module where it is located is compiled with the -globcurs option.

The FREE statement may be followed by the identifier of the prepared statement, which will free the memory allocated to the prepared statement. After you have deallocated memory from the prepared statement, you cannot reference the identifier by a cursor or by the EXECUTE statement.

If you declare a new prepared statement with the existing identifier the old contents of the prepared statement will be replaced by the new one.

The Text of the Prepared Statement

The text of the prepared statement is a quoted character string or a variable of character data type that returns such string. It immediately follows the PREPARE statement. There are some restrictions which should be considered when preparing a statement:

- The text of the prepared statement can contain only SQL or SPL statements that can be prepared. The 4GL, C and C++ statements are **not** allowed. SQL statements that cannot be prepared cannot be included into the text either.
- If a prepared statement consists of more than one SQL statement, such statements should be separated by semicolons
- The hash symbol (#) **cannot** be used as the comment indicator. Only the double hyphen symbol (--) and double braces ({{}}) are allowed as the comment symbols
- The SQL identifiers (names of tables and columns) are the only possible identifiers that can be used in a prepared statement. It cannot contain 4GL variables, e.g. an INTO clause which contains a 4GL variable cannot be used in the prepared statement



- A question mark (?) can be included into the text to specify the values that should be supplied by the user at runtime. This must not be an SQL identifier.

```
PREPARE p_country FROM "SELECT * FROM qxt_country", "WHERE country_name=?"
```

In the example above, the question mark specifies that the criterion for selection should be supplied by the user.

Preparing the SELECT Statement

The SELECT statement can be used directly in the 4GL code, or it may be prepared. If the prepared SELECT statement contains the INTO TEMP clause, it can only be executed by means of the EXECUTE statement. If the SELECT statement does not contain the INTO TEMP clause, the cursor can retrieve the rows selected by such statement with the help of the FOREACH statement or the OPEN and FETCH statements.

	Note: The FOREACH statement cannot be used with the prepared SELECT statement that contains a question mark
---	--

The prepared SELECT statement can include the FOR UPDATE clause. Such SELECT statements are usually used together with the FOR UPDATE cursor declaration. If the example below, the SELECT statement is prepared from a variable:

```
DEFINE sql_stmt CHAR(2032)
LET sql_stmt = "SELECT * FROM account_info", "WHERE acc_id BETWEEN ? and ?
", "FOR UPDATE"
PREPARE p_act_arr FROM sql_stmt
DECLARE acc_cur CURSOR FOR p_act_arr
OPEN acc_cur USING low_acc, high_acc
```

SQL Statements that Must be Prepared

Some SQL statements cannot be embedded into the 4GL code, they need to be prepared or they must be used within the SQL ... END SQL statement. These statements must be prepared, if you want to use them in a 4GL program:

ALTER FRAGMENT	SET CONSTRAINT
ALTER OPTICAL CLUSTER	SET DATABASE OBJECT MODE
CREATE EXTERNAL TABLE	SET DATASKIP
CREATE OPTICAL CLUSTER	SET DEBUG FILE TO
CREATE ROLE	SET LOG
CREATE SCHEMA	SET MOUNTING TIMEOUT
CREATE TRIGGER	SET OPTIMIZATION
DROP OPTICAL CLUSTER	SET PDQPRIORITY
DROP PROCEDURE	SET PLOAD FILE
DROP ROLE	SET RESIDENCY



DROP TRIGGER	SET ROLE
EXECUTE PROCEDURE	SET SCHEDULE LEVEL
GRANT FRAGMENT	SET SESSION AUTHORIZATION
RELEASE	SET TRANSACTION
RENAME DATABASE	SET TRANSACTION MODE
RESERVE	START VIOLATIONS TABLE
REVOKE FRAGMENT	STOP VIOLATIONS TABLE

SQL Statements that Can Be Embedded

There are a number of SQL statements that can be directly used in a 4GL code and do not require being prepared. Most of those SQL statements which cannot be embedded can be prepared or used within the SQL ... END SQL statement.

You do not need to prepare the following statements, you can use them directly in the 4GL code:

ALTER TABLE	CREATE SYNONYM
CREATE INDEX	INSERT INTO
CREATE TABLE	DROP TABLE
CREATE SCHEMA	DROP VIEW
GRANT	UPDATE STATISTICS
REVOKE	

SQL Statements that Cannot be Prepared

The following SQL statements cannot be used in the PREPARE statement:

ALLOCATE COLLECTION	FREE
ALLOCATE DESCRIPTOR	GET DESCRIPTOR
ALLOCATE ROW	GET DIAGNOSTICS
CHECK TABLE	INFO
CLOSE	LOAD
CONNECT	OPEN
CREATE PROCEDURE FROM	OUTPUT
DEALLOCATE COLLECTION	PREPARE
DEALLOCATE DESCRIPTOR	PUT
DEALLOCATE ROW	REPAIR TABLE
DECLARE	SET AUTOFREE
DESCRIBE	SET CONNECTION
DISCONNECT	SET DEFERRED_PREPARE
EXECUTE	SET DESCRIPTOR
EXECUTE IMMEDIATE	UNLOAD
FETCH	WHENEVER
FLUSH	

The following statements cannot be used in the PREPARE statement that consists of several SQL statements separated by semicolons: DATABASE, CLOSE DATABASE, CREATE DATABASE, DROP DATABASE, START DATA BASE, SELECT.



	Note: You can use only the SELECT INTO TEMP statement (not other types of the SELECT statement) in the PREPARE statement that contains other SQL statements
---	--

Such statements as CONNECT, DISCONNECT, and SET CONNECTION cannot be used in a PREPARE statement that includes a number of other statements.

Executing Prepared Statements

To create and then execute a stored procedure using 4GL, follow these steps:

1. Type the CREATE PROCEDURE statement in a separate source file. Use the SPL statements to define the procedure
2. Embed the statement CREATE PROCEDURE FROM *file-name* into a 4GL source file, it must reference the file created in step 1
3. Prepare the EXECUTE PROCEDURE statement (which executes the procedure created and referenced in steps 1 and 2) by means of the PREPARE statement
4. Use the EXECUTE PROCEDURE statement to execute the statement prepared in step 3

The SPL is not a part of Querix 4GL language, SPL statements cannot be embedded directly into the 4GL code. It can be used only by means of the CREATE PROCEDURE FROM *file-name* statement.

You can execute a stored procedure also by referencing it within a SQL expression. Such implicit references do not need to be prepared.

Parameters of the PREPARE Statement

The values that are used in the text of a PREPARE statement are called parameters. They can be supplied when the statement is prepared (if they are known), or they can be supplied at the runtime by the user (if they are unknown at the moment when the statement is prepared).

The PREPARE Statement with Known Parameters

If all the information needed for processing the prepared statement is known at the moment of its preparation, it can be included into the PREPARE statement. These parameters can be included into the prepared text as literals or as variables. You can also include a variable that requires the user input as a parameter:

```
DEFINE us_inp LIKE client_info.lname
PROMPT "Enter the last name of the client: " FOR us_inp
PREPARE client_search FROM
"SELECT * FROM client_info",
"WHERE lname = '", u_po, "'"
DECLARE cl_sear CURSOR FOR client_search
```



The PREPARE Statement with Unknown Parameters

If the different values are supposed to be inserted into the prepared statement each time it is executed, such parameters are unknown. The question mark (?) can be used in places where the unknown parameters should be. The values for these parameters will be supplied to the PREPARE statement when it is executed.

```
PREPARE stat FROM "INSERT INTO order_info VALUES (?,?,?,?,?)"
```

A question mark can be used only to supply a value for an expression, it cannot be used to substitute SQL identifiers.

You can use the USING clause when you execute the prepared statements either with the help of the EXECUTE, FOREACH or OPEN statement.

SQL Identifiers Used in Prepared Statements

The SQL identifiers cannot be replaced by question marks in prepared statements. The identifiers must be included into the text of the prepared statement at the moment when it is prepared. However, if the statements are unknown at the moment of preparation, you can create a construct which will retrieve the identifiers from the user input:

```
DEFINE db_col CHAR(25),
db_col_val CHAR(40),
ins_stmnt CHAR(100)

PROMPT "Enter column name: " FOR db_col
LET del_str = "INSERT INTO order WHERE ",
db_col CLIPPED, " = ?"
PREPARE inser_3 FROM ins_stmnt
PROMPT "Enter search value in column ",column_name, ":""
FOR column_value
EXECUTE de4 USING column_value
```

Preparing Multiple SQL Statements

Several SQL statements can be included into one PREPARE statement. When such prepared multi-statement is executed, it is processed as a single unit and the component statements are executed one by one.

A statement that depends on the execution of the statements which precede it cannot be used in a PREPARE statement. E.g. one PREPARE statement cannot include the statements that create a table and after that insert data into it. These statements must be in different PREPARE statements. It is also advisable not to place the BEGIN WORK and COMMIT WORK statements within a prepared multi-statement.

4GL returns error status information on the first statement of a multi-statement. 4GL does not specify which of the statements included into it has produced the error. The following example is an example of a prepared multi-statement:

```
DATABASE cms
```



```
MAIN
DEFINE company_update RECORD
comp_name_new LIKE company.comp_name,
comp_name_old LIKE company.comp_name
END RECORD
sql_stmt_m CHAR(1000)
PROMPT "Enter new company name: " FOR company_update.comp_name_new
PROMPT "Enter old company name: " FOR company_update.comp_name_old
LET sql_stmt_m = "UPDATE company SET company.comp_name = ? WHERE
company.comp_name = ? ;",
"UPDATE logo_info SET menu_code = ? WHERE manu_code = ? ;",
"UPDATE order_info SET manu_code = ? WHERE manu_code = ? ;"
PREPARE st_mult FROM sql_stmt_m
EXECUTE st_mult
    USING company_update.comp_name_new, company_update.comp_name_old,
manu_code_new, manu_code_old, manu_code_new, manu_code_old
END MAIN
```

Runtime Errors in the Multistatements

If an error occurs during the execution of one of the component statements of a multi-statement, the execution of the subsequent statements within the same multi-statement is cancelled. 4GL will not test the subsequent statements for errors until the previous error is fixed.

If the multi-statement contains more than one error, the information about subsequent errors is not reflected in the error status information. Any error returned by a database server terminates the execution of the prepared multi-statement, without regard to the severity of this error.

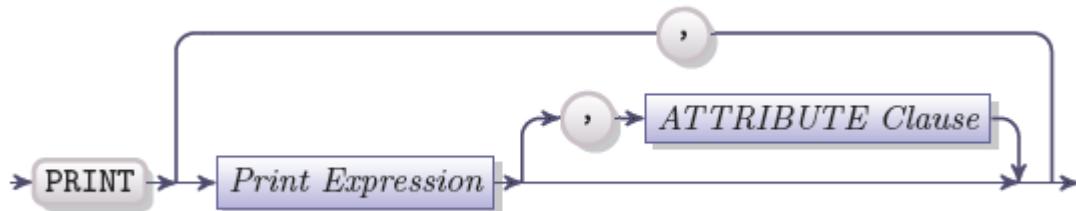
Usage of the Prepared Statements

The prepared statements can be used to improve the efficiency of your 4GL program. For example, if you place use the INSERT or UPDATE statement in a WHILE loop, the statement is processed each time the loop is executed. This leads to overhead and 4GL spends much time for processing these loops. If you place the prepared INSERT or UPDATE statement outside the loop and then execute it within the loop, the statement will be processed only once and then the result of its execution will be used in the WHILE loop. This will speed up the execution of the loop and eliminate the overhead.



PRINT

The PRINT statement is used to perform output from a report. It can be used only within the REPORT program block.



Element	Description
Print Expression	A 4GL expression the value of which is sent to a report or a print operator
ATTRIBUTE Clause	The display attributes of the value sent to a report

You cannot specify an additional output for the TEXT variable or a file in the same PRINT statement. The values of BYTE data type cannot be displayed by means of the PRINT statement. The PRINT statement sends the output to the destination specified in the REPORT TO clause of the OUTPUT section of the REPORT program block, or in the TO clause of the START REPORT statement. If the destination is not specified, the output is printed on screen. Below is the example which includes both quoted character strings and variables used in PRINT statements.

```
FORMAT
FIRST PAGE HEADER
    PRINT COLUMN 45, "Items List"
    SKIP 1 LINES
    PRINT "Sorted by ", ord_opt
    SKIP 1 LINES
    PRINT "NUMBER", COLUMN 18, "CLIENT", COLUMN 46, "AMOUNT",
    COLUMN 57, "SUM"
    SKIP 1 LINE
PAGE HEADER
    PRINT "NUMBER", COLUMN 18, "CLIENT", COLUMN 46, "AMOUNT",
    COLUMN 57, "SUM", ATTRIBUTE(RED)
    SKIP 1 LINE
ON EVERY ROW
    PRINT order_id USING "###&", COLUMN 18, comp_name CLIPPED,
    1 SPACE, COLUMN 46 it_amount CLIPPED, COLUMN 57, tot_sum CLIPPED
```



The Print Expression

The Print expression is an expression which returns a value sent to the report. One PRINT statement can have more than one print expression, they should be separated by commas. It can be one of the expressions listed below:

- 4GL expression – the value returned by the expression is printed.
- COLUMN *integer* – where *integer* returns a positive integer specifying the character position left offset, it cannot be a greater value than the difference (right margin - left margin).
- PAGENO – returns the number of the current report page
- LINENO – returns the number of the currently printing line
- *integer* SPACE / *integer* SPACES – where *integer* returns a positive integer specifying the number of whitespaces between the last symbol of the previous value and the first character of the next one
- Aggregate report functions – functions that return the total value resulting from a number of input records within a report
- *character_expression* WORDWRAP – where *character_expression* is a long character string or a TEXT variable which will be automatically wrapped onto successive lines
- TEXT *variable* – where *variable* is the text variable the value of which will be printed
- FILE "filename" – where *filename* is the name of a text file which must be printed

ATTRIBUTE Clause

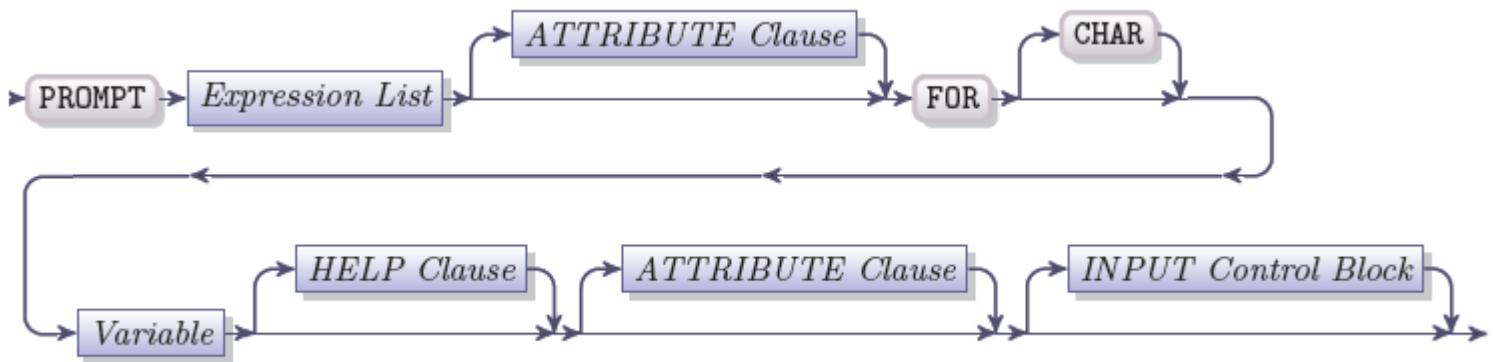
Each print record can have its own ATTRIBUTE clause which specifies the display attribute for the printed value. This clause supports all the standard display attributes of colour and intensity described for the common [ATTRIBUTE clause](#).

For more detailed information, see the [PRINT section](#) in the REPORT program block description.



PROMPT

The PROMPT statement assigns values supplied by the users to the program variables.



Element	Description
Expression List	Expression List which returns the message displayed by the prompt statement
ATTRIBUTE Clause	Display attributes applied to the displayed message and to the entered value
Variable	The variable to which the entered value is assigned
HELP Clause	The clause that specifies the number of the help message
INPUT Control block	The clause that controls the input of the prompted value

The PROMPT statement must be followed by the message that is displayed on the PROMPT line and that informs the user what kind of data are expected. The message may be represented by a quoted character string, or by a variable of character data type. When the PROMPT statement is executed, 4GL waits for the user input. After the value is entered, it is assigned to the variable specified in the FOR clause, unless the typed value coincides with the key-name specified in the ON KEY clause.

The following actions are performed when 4GL encounters a PROMPT statement:

1. If any variables are present in the prompt text, they are replaced with their current values.
2. The values, and quoted character strings are concatenated into a single prompt message (the total length of such message must not exceed 80 bytes)
3. The prompt message is displayed on the PROMPT LINE (which is the first line of a 4GL window or 4GL screen by default)
4. 4GL waits for the user input
5. The entered value is assigned to the variable specified in the FOR clause.

The prompt field remains active and the prompt message remains visible until the input is performed.

The PROMPT statement must include such components:

- PROMPT string
- FOR clause



There are also optional components that may follow the required components:

- ATTRIBUTE clause
- HELP clause
- PROMPT input control clause
 - BEFORE PROMPT clause
 - ON KEY and ON ACTION clauses
 - AFTER PROMPT clause

If at least one PROMPT input control clause is present within the PROMPT statement, the END PROMPT statement is required to mark the end of the PROMPT statement.

The PROMPT input control clauses can be listed in any order within the PROMPT statement, but they will be executed in the order in which they are listed above. However, all the other optional clauses must be placed in the order specified by the diagram, otherwise a compile-time error may occur.

The PROMPT String

The PROMPT string is a quoted character string, a variable of any simple or structured data type or a combination of the strings and variables. The prompt string must immediately follow the PROMPT keyword. It is displayed as a prompt message on the PROMPT LINE, which is the first line of the screen or window. Its default position can be changed by means of the OPTIONS statement. If the prompt message is longer than the PROMPT LINE, a runtime error occurs.

The FOR Clause

The FOR clause consists of the FOR keyword and the name of the variable which is to store the value which the user enters when prompted. When the user types the value and presses the RETURN key, the value is saved in the variable. The variable that follows the FOR keyword can be of any simple or structured data type. It can store blank spaces, if the variable is of character data type. If the value entered into the PROMPT field is not compatible with the variable, an error will occur.

```
PROMPT "Enter the first letter of the last name: " FOR f_let
      CALL search(f_let)
```

The CHAR Keyword

You can include the CHAR clause between the FOR keyword and the variable. The input of a single character will be accepted at once, the RETURN key need not to be pressed. You cannot type more than one character in such PROMPT field, the single character will be accepted as the value of the PROMPT variable.

If the CHAR keyword is present, neither HELP clause nor ON KEY or ON ACTION clauses can be executed. However, the BEFORE PROMPT, AFTER PROMPT and ATTRIBUTE clauses will be executed normally.

The HELP Clause

The HELP clause specifies the number of the help message that is associated with the PROMPT message.

— **HELP** → *Integer Expression* →



Element	Description
Integer Expression	A 4GL expression that returns a positive integer greater than 0 and less than 32,767.

The HELP keyword can be followed either by a literal integer or by a 4GL expression or a variable that returns a positive integer. The help message is invoked when the user presses the help key. By default CONTROL-W is the help key. The default settings can be changed in the OPTIONS statement.

The help file which contains the help message referenced in the HELP clause should be specified in the HELP FILE clause of the OPTIONS statement. A runtime error occurs if:

- the help file cannot be opened
- the help message number referenced in the HELP clause is absent from the help file
- the number of the help message specified in the HELP clause is smaller than -32,767 or larger than 32,767

The ATTRIBUTE Clause

The ATTRIBUTE clause of the PROMPT statement has the structure of the common [ATTRIBUTE clause](#). It specifies the display attributes applied to the PROMPT message. These attributes override the settings of any other preceding ATTRIBUTE clause (of OPTIONS or OPEN WINDOW statements).

Two ATTRIBUTE clauses can be specified within one PROMPT statement:

- The first ATTRIBUTE clause specifies the display attributes for the PROMPT string (the default is NORMAL) – it is placed after the PROMPT message and before the FOR clause.
- The second ATTRIBUTE clause specifies the display attributes for the prompt field – the area for user input (the default is REVERSE) – it is placed immediately after the FOR clause

```
PROMPT "Enter the item number " ATTRIBUTE (GREEN, REVERSED) FOR it_num
        ATTRIBUTE (GREEN)
```

Each attribute clause is independent and affects only the part of the PROMPT statement it is assigned to.

The CENTURY Attribute

If the variable of the FOR clause is of the DATE or DATETIME data type, the ATTRIBUTE clause can include the CENTURY attribute. The semantics of this attribute is the same as the CENTURY attribute of a form field.

The CENTURY attribute is used to define how a date entered into the PROMPT field will be expanded, if only one or two digits are entered for year value. It supports the settings of the DBCENTURY environment variable.

The CENTURY attribute has the following structure:

CENTURY =	"R"	Adds two first digits of the current year	The letters can be either lower case or upper case
	"C"	Expands the two digits to the year closest to the current date (either in present, past, or future)	
	"F"	Expands the two digits to the nearest	



"P"	year in future Expands the two digits to the nearest year in past
-----	--

If you specify anything for this attribute other than the four letters listed in the table above, an error will occur.

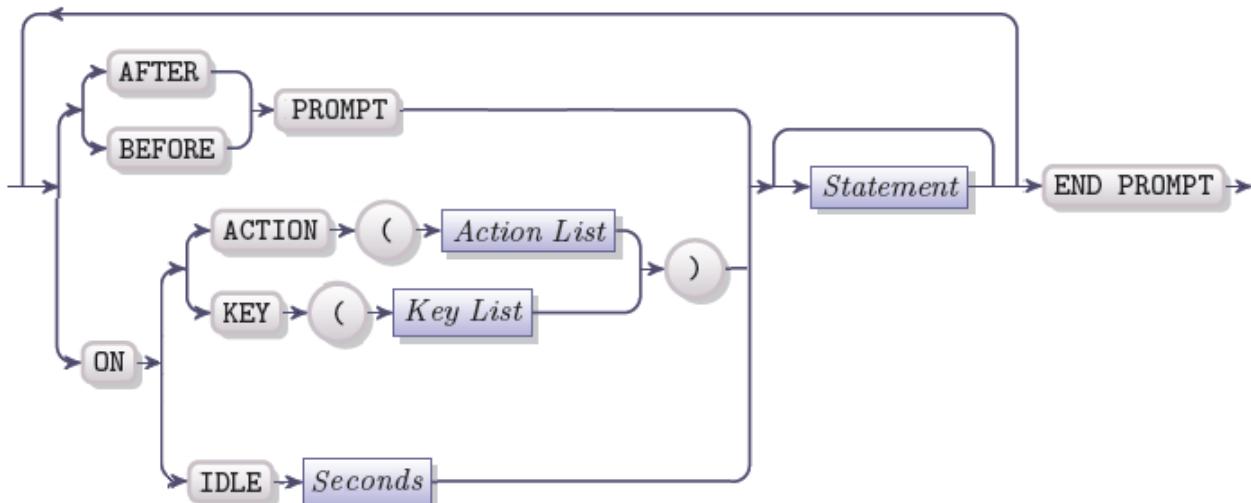
If the settings of the DBCENTURY environment variable differ from the settings of the CENTURY attribute, the CENTURY attribute overrides the settings of the environment variable.

```
PROMPT "Enter the date of order" ATTRIBUTE (REVERSE, GREEN) FOR ord_date
ATTRIBUTE (BLUE, CENTURY = "P")
```

The above example displays the prompt message in green and reversed and the user input is displayed in blue colour. If the user supplies the date with only two digits for the year, the date will be expanded to the closest date in the past that matches the entered data. E.g. if user enters 12/12/98 – the displayed date will be 12/12/1998.

The PROMPT Input Control Clauses

These clauses specify the actions 4GL should undertake before or after executing the PROMPT statement or if the user presses a key or a button on the screen form.



Element	Description
Event List	PROMPT keyword
Action List	A list that consists of one to four actions separated by commas and enclosed in quotation marks
Key List	A list that consists of one to four key names separated by commas
Seconds	An integer number specifying the timeout period in seconds
Statement	The statement block of the INPUT control clause

If a PROMPT statement contains at least one PROMPT input control clause, the END PROMPT keywords must mark the end of the PROMPT statement.



The Statement Block

The statement block following each PROMPT control clause must include at least one executable statement. It can include 4GL or SQL statements. Unlike other statements that have the input control clause, the PROMPT statement does not support the EXIT PROMPT or CONTINUE PROMPT statements, while it is automatically completed after the input is performed or a key or a button is pressed and cannot be continued afterwards.

The BEFORE PROMPT Clause

The statements in the BEFORE PROMPT clause are executed before the user is allowed to enter data into the PROMPT field. This clause can be used to display messages that inform the user what should be entered into the PROMPT field, or about the help key, etc.

The PROMPT statement can have only one BEFORE PROMPT clause. If this clause is present, the END PROMPT keywords are required.

The ON KEY Clause

The ON KEY clauses can be included into the PROMPT statement to specify the behaviour of the keys pressed while the PROMPT statement is executed and the user input is performed. The keys can be assigned to form widgets and the actions will be triggered when such widget is pressed. However, if a key combination is not assigned to any widget, it can be activated with the help of the keyboard. One PROMPT statement can contain any number of the ON KEY clauses, they can be placed in any order.

The ON KEY clause can use one or several of the following keys separated by commas enclosed in parentheses to specify the key to which the other statements in the ON KEY clause refer:

ESC/ESCAPE	ACCEPT
NEXT/ NEXTPAGE	PREVIOUS/PREVPAGE
INSERT	DELETE
RETURN	TAB
INTERRUPT	HELP
LEFT	RIGHT
UP	DOWN
F1 – F256	CONTROL- <i>char</i>

Any character can be used as *char* in the combination CONTROL-*char* except the following characters: A, D, H, I, J, K, L, M, R, and X. The key names can be entered either in lower case or in upper case letters.

Any 4GL or SQL statements can be used in the ON KEY clause. These statements are executed when the key specified in the corresponding ON KEY clause is pressed.

When an ON KEY clause within the PROMPT statement is activated by pressing the corresponding key, the following actions are performed:

1. The program control is passed to the corresponding ON KEY clause
2. The statements of the ON KEY clause are executed
3. The control is passed to the first statement that follows the END PROMPT keywords.

If the user presses the key specified in the ON KEY clause, this key is not typed in the PROMPT field. As the control does not return to the PROMPT statement after the ON KEY clause is executed, the value entered



into the PROMPT field is assigned to the variable in the FOR clause. If the key is pressed before any value has been entered into the PROMPT field, the variable remains undetermined.

The value entered by the user can be changed by assigning a new value to the variable used in the FOR clause by means of the statements of an ON KEY clause.

```
PROMPT "Enter the date of order" ATTRIBUTE (REVERSE, GREEN) FOR ord_date
ATTRIBUTE (BLUE, CENTURY = "C")
ON KEY (F5)
LET ord_date = TODAY
END PROMPT
```

Some keys require special attention if used in the ON KEY clause:

Key	Usage Features
ESC/ESCAPE	If you want to use this key in ON KEY clause, you must specify another key as the Accept key in the OPTIONS statement, because ESCAPE is the Accept key by default
INTERRUPT	The DEFER INTERRUPT statement must be executed in order that this key could be used in the ON KEY clause. On pressing the Interrupt key the corresponding ON KEY clause is executed, int_flag is set to non-zero value
QUIT	The DEFER QUIT statement must be executed in order that this key could be used in the ON KEY clause. On pressing the QUIT key the corresponding ON KEY clause is executed and int_flag is set to non-zero value
CTRL-char (A, D, H, K, L, R, X)	4GL reserves these keys for field editing and they should not be used in the ON KEY clauses
CTRL-char (I, J, M)	These key combinations by default mean TAB, NEWLINE and RETURN. If they are used in the ON KEY clause, they cannot perform their default functions while the OK KEY clause is functional. Thus, if you use these keys in an ON KEY clause, the period of time during which this clause is functional should be restricted.

Some restrictions in key usage might be applied depending on your operational system. Many systems use such key combinations as CONTROL-C, CONTROL-Q, and CONTROL-S for the Interrupt, XON, and XOFF signals.

The ON ACTION Clauses

The actions which you have assigned to the buttons in the form specification file can be used in an ON ACTION clause. The ON ACTION clauses are treated by 4GL very much like the ON KEY clauses. 4GL executes the statements specified in it, when the user presses a button or other widget on the form to which the corresponding action is assigned. Whereas an ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard, an ON ACTION clause is activated with the help of a form widget.

An event (action) can be assigned to the following widgets: button, radio button, check box, function field, and hotlink. It must be entered into the corresponding field in the Graphical Form Editor without quotation marks and white spaces. In the text form editor it must be enclosed into quotation marks:

```
CONFIG = "action {Name of the Button}"
```



The actions in an ON ACTION clause must be represented by a character string enclosed both in quotation marks and in parentheses and it must be the same as the name of the action in the form specification file:

```
ON ACTION ("action")
```

Lycia II also supports an in-built action named "*doubleclick*". The statements specified in the ON ACTION ("doubleclick") control block are invoked when the user performs a double click of the left mouse button at any place of the application window.

```
ON ACTION ("doubleclick")
```

It is also possible to set different doubleclick actions for different form widgets. To do this, one should add the INFIELD clause and specify the field or fields, double clicking on which should invoke the statements in the ON ACTION block:

```
ON ACTION ("doubleclick") INFIELD (f001)
```

There can be any number of the ON ACTION clauses in a PROMPT statement. Up to four actions enclosed in quotation marks and separated by commas may be included into one ON ACTION clause.

As well as with the ON KEY clause, any 4GL or SQL statements can be included into an ON ACTION clause. The statements of a corresponding ON ACTION clause are executed when the widget within a screen form to which the action is assigned is pressed.

If a PROMPT statement contains at least one ON KEY clause or ON ACTION clause, the END PROMPT keywords are required.

When an ON ACTION clause within the PROMPT statement is activated by pressing the corresponding form widget, the following actions are performed:

1. The program control is passed to the corresponding ON ACTION clause
2. The statements of the ON ACTION clause are executed
3. The control is passed to the first statement that follows the END PROMPT keywords.

As the control does not return to the PROMPT statement after the ON ACTION clause is executed, the value entered into the PROMPT field is assigned to the variable in the FOR clause. If the form widget is pressed before any value has been entered into the PROMPT field, the variable remains undetermined.

	Note: Please note that only one key or one button (specified in an ON KEY or ON ACTION clause) can be pressed during the PROMPT statement, even if the PROMPT statement contains more than one ON KEY or ON ACTION clauses. Because unlike the INPUT and INPUT ARRAY statement the control is not returned to the PROMPT statement afterwards.
--	---

The example below represents an ON ACTION clause which is triggered when the button is pressed which has the "*default_date*" event assigned.

```
PROMPT "Enter the date of order" ATTRIBUTE (REVERSE, GREEN) FOR ord_date  
ATTRIBUTE (BLUE, CENTURY = "C")
```



```
ON ACTION ("default_date")
LET ord_date = "01/01/2000"
END PROMPT
```

The ON IDLE Clauses



Element	Description
Seconds	An integer value specifying the time period in seconds

The ON IDLE clause is used to specify the actions that will be taken if the user performs no actions during the specified period of time.

The *seconds* parameter should be represented by an integer value or a variable which contains such a value. If the value is specified as zero, the input timeout is disabled.

It is advised that you specify the *seconds* parameter as a relatively long period of time (more than 30 seconds), because shorter delays may be caused by some external situations that distract the user from the application, so, the control block will be executed inappropriately:

```
PROMPT "Enter your name" FOR nme
ON IDLE 120
CALL fgl_message_box("You've been out for 2 minutes")
END INPUT
```

The AFTER PROMPT Clause

The statements in the AFTER PROMPT clause are executed after the user presses the RETURN key, but before the value entered by the user is assigned to the variable. A PROMPT statement can have only one AFTER PROMPT clause.

The AFTER PROMPT clause is executed after the user presses the RETURN key or Accept key or after any key has been pressed, if the PROMPT statement contains the CHAR keyword. The AFTER PROMPT clause is ignored, if a user pressed a key specified in the ON KEY clause or a form widget specified in the ON ACTION clause.

The END PROMPT Keywords

The END PROMPT keywords specify the end of the PROMPT statement. They are required, if the PROMPT statement contains at least one PROMPT input control clause. It should be placed after the last PROMPT input control clause.

The PROMPT LINE

The PROMPT message is displayed on the prompt line, by default it is the first line of a 4GL window or screen. The default settings can be changed by specifying the new PROMPT LINE in:

- ATTRIBUTE CLAUSE of the OPEN WINDOW statement
- PROMPT LINE clause of the OPTIONS statement



The PROMPT LINE keywords can be followed by a literal positive integer, an integer expression or the keywords used to specify the position of the reserved lines (FIRST, LAST) with or without literal integers. If you specify the position for the PROMPT LINE that exceeds the existing lines of the current 4GL window, it will be displayed on the first line. As sometimes the number of lines in different windows differs, it is better to define a relative position for the PROMPT LINE. E.g.:

```
OPTIONS PROMPT LINE LAST-3
```



READ

The READ statement is used to read from the file and to record the read values into the corresponding program variables.



Element	Description
Descriptor	An integer variable storing the file descriptor
Variable List	A list of variables separated by commas that will accept the values read from the file

The READ statement reads from the file previously opened by the OPEN FILE statement. While the file is open, each new READ statement reads the file further from the location the previous READ statement ended reading. Closing and opening the file again will reset the reading offset and the reading will start at the beginning of the file.

There is no markers that define the end of the read file. The status variable does not change its value from 0, if a READ statement tries to read a non-existent value and goes beyond the end of file. So if you try to read a file in a loop checking that the status variable is 0, the loop will never end. The value into which a non-existent value tries to be read will not change its contents and will not be assigner NULL as the result.

Variable list

The variables into which the values are read should be compatible with these values, otherwise a conversion error will occur. Each variable will accept a value between the specified delimiters, that depend on the file format and the delimiters set.

When reading from a "load" file format, each variable will contain a value between the delimiters (| by default). The escape symbol for the delimiter is backslash (\). The newline characters will be ignored. E.g. the following unl file contains 4 values that will be recorded in to the variables by the READ statement:

```
John|Smith|
Sam|Vimes|
...
READ FROM file_dscr INTO fname1, lname1, fname2, lname2
```

When reading a csv file, each variable will contain a value between commas, because comma is the default separator for csv file format. The newline characters will be ignored.

```
John,Smith,
Sam,Vimes,
...

```



```
READ FROM file_dscr INTO fname1, lname1, fname2, lname2
```

When reading from a text file, each variable will contain a new line, since the newline character works as the default separator:

```
John Smith - line 1  
Sam Vimes - line 2
```

```
...
```

```
READ FROM file_dscr INTO line1, line2
```

It is possible to use records and arrays as the variables in the INTO clause:

```
READ FROM file_dscr INTO my_rec.*  
READ FROM file_dscr INTO my_array
```



REPORT

The REPORT statement declares and defines the REPORT program block. This block contains instructions that deal with the format of a 4GL report.



Element	Description
Report Name	The name you want to assign to the report
Argument List	Formal arguments passed to the report
REPORT Sections	Sections used to format the report output and declare local variables

For more detailed information about the REPORT program block, see the chapter that concerns the creation and usage of the REPORT - "[CHAPTER 4: Reports](#)".

The REPORT statement defines the REPORT program block, which is similar to the FUNCTION program block. It can be executed from the MAIN program block or from a FUNCTION program block. The REPORT statement itself cannot be used within some other program block, it cannot also appear within the other REPORT program block.

To create a report, do the following:

1. Create the REPORT program block which formats the data passed to the report
2. Use the report driver in the program routine which invokes and executes the report to process it

A report driver is typically used in a loop (WHILE, FOR or FOREACH) and it consists of the following statements, which process the report:

- The START REPORT statement is used to invoke a report
- The OUTPUT TO REPORT statement specifies which information should be sent to a report
- The FINISH REPORT statement is used to finish the processing of a report
- The TERMINATE REPORT statement is used to terminate the processing of a report; it is usually used within a conditional statement

The REPORT program block is not iterative. If you use the START REPORT statement for a report that is already running, the record will be restarted and the output might be unexpected.

If the OUTPUT TO REPORT statement is not executed, the report is not issued even if you include the START REPORT and FINISH REPORT in your program.

The Report Prototype

The report name must follow the REPORT keyword. The name of the report must be unique among other reports within the same program.

A report may have formal arguments. The list of such arguments must be enclosed in parentheses and should be separated by commas. The formal arguments should be declared with the help of the DEFINE

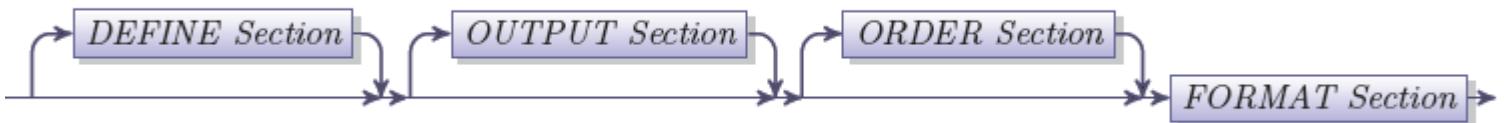


statement at the beginning of the REPORT program block. They are used to pass values from the calling routine to the report. The formal arguments can be of any simple data type and of the RECORD data type (with the .* notation after the name of the record). The variables of ARRAY data type or records with program arrays as members cannot be used as formal arguments.

The values are passed to the formal arguments from the actual argument list of the OUTPUT TO RECORD statement. They must match the formal arguments in order, number and be of compatible data types. The names of the formal and actual arguments do not have to match.

The REPORT Program Block

The REPORT program block must include the FORMAT section, it can also contain the DEFINE, OUTPUT, and ORDER BY statements.



Element	Description
DEFINE Section	Here the local variables are declared
OUTPUT Section	Here the dimensions and other attributes of a report page are set
ORDER Section	Here the rules for sorting the input records passed to the report are set
FORMAT Section	Here the format in which the report information is sent to the report destination is specified

The detailed information about the report sections can be found in "[CHAPTER 4: Reports](#)".

The DEFINE statement must include the declarations of all the formal arguments and variables local to the REPORT program block. The DEFINE statement must immediately follow the list of the formal arguments. The variables declared by means of the DEFINE statement override any other variables of greater scope of reference, if they have the same name. Variables local to a report cannot be referenced from outside the report. These variables do not retain values between the invocations of the REPORT program block.

The following items must be included into the list of formal arguments:

- The values for each row sent to the report if:
 - the ORDER BY clause is included into the report
 - the GROUP PERCENT(*) function is included into the report
 - a global aggregate function is used anywhere in the report except in the LAST ROW clause
 - the FORMAT EVERY ROW is specified as the default format
- The values for the variables referenced in the following clauses:
 - the BEFORE GROUP OF clause
 - the AFTER GROUP OF clause

The Two-Pass Reports

A two-pass report creates a temporary table. A temporary table is created if:



- An ORDER BY clause with the EXTERNAL keyword is included into the report
- The GROUP PERCENT(*) function is included into the report
- An aggregate function without GROUP keyword is included into any control clause of the REPORT other than ON LAST ROW

The temporary tables are deleted after the FINISH REPORT statement uses them to calculate the global aggregates.

To create a two-pass report, the program must have access to a database at the time when the report is run. Thus the current database must be defined. To see details about how to specify the current database, see the DATABASE statement description.

You must specify the default database in the REPORT program block to be able to use the DEFINE...LIKE statement to declare local report variables like the table columns. To specify the default database, include the DATABASE statement in the same module where the REPORT program block is located, before the first program block within that module. To see details on how to specify the default database, see the DATABASE statement description.

When a two-pass report is running, do not change or close the current database, otherwise an error will occur. The current database is required when the two-pass report is running, even if it does not retrieve any records from a database, because it uses the database to store temporary tables.

The EXIT REPORT Keywords

The EXIT REPORT statement can appear in the FORMAT clause. It terminates the report as well as the TERMINATE REPORT statement. The TERMINATE REPORT appears in the routine that executes the report, in the report driver. The EXIT REPORT statement must appear only within the REPORT program block. The RETURN statement cannot be used instead of the EXIT REPORT statement, an error will occur, if the RETURN statement is placed within the REPORT program block.

When 4GL encounters the EXIT REPORT statement, it performs the following actions:

- Terminates the processing of the currently run report
- Deletes temporary files and tables which were created while processing the report

The END REPORT Keywords

The END REPORT keywords specify the end of the REPORT program block. A REPORT program block can have only one END REPORT statement.

Below is an example of a report driver and a REPORT program block:

```
DECLARE curl_ord CURSOR FOR SELECT * FROM order_info
START REPORT ord_inf
FOREACH curl_ord INTO ord_rec.*
OUTPUT TO REPORT ord_inf (ord_rec.*)
END FOREACH
FINISH REPORT ord_inf
...
REPORT ord_inf (p_ord_rec)
```

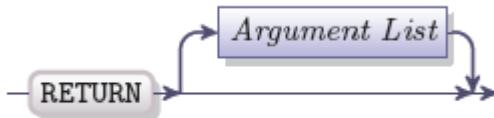


```
DEFINE x RECORD LIKE order_info.*  
FORMAT EVERY ROW  
END REPORT
```



RETURN

The RETURN statement is used only within the FUNCTION program block. It returns control from a function to the calling routine and also returns values from the function.



Element	Description
Argument List	A list of values returned by the function to the calling routine

The RETURN statement usually appears at the end of the FUNCTION program block, before END FUNCTION keywords, because it returns control from the function to the calling routine and all the statements placed between the RETURN statement and the END FUNCTION keywords will be ignored. It can be also used in a conditional statement within the FUNCTION program block in order to return control from the function to the calling routine under some conditions.

The RETURN statement can be used either with or without values:

- Without values the RETURN statement is used to control the course of the program execution (to leave the function and to return the program control to the calling routine under some conditions)
- With a list of values The RETURN statement is used to pass the values to the calling routine. It also returns the program control to the calling routine.

The RETURN statement is optional and it can be absent from a function. If there is no RETURN statement, a function will be terminated, when 4GL encounters the END FUNCTION keywords and the program control will be passed to the calling routine.

If you use the RETURN statement within a program block other than FUNCTION program block (MAIN or REPORT), an error will occur.

Returned Values

The RETURN statement can be used with the list of returned values. The values must immediately follow the RETURN keyword, they must be separated by commas. The THRU/THROUGH keyword and the .* notation can be used to specify a part of a program record or a whole record.

To invoke a function, do the following:

- specify the function name in the CALL statement (if the function does not return anything)
- specify the function name in the CALL statement with the RETURNING clause (if a function returns more than one value)
- use the function as an operand within a 4GL expression (if a function returns a single value)



Returned Data Types

If a function returns values, the number and order of the arguments in the RETURNING clause of the CALL statement must match the returned values in the RETURN statement in number and order. Their data types must be the same or compatible.

If a function returns exactly one value and is invoked implicitly as an operand of a 4GL expression, the data type of the returned value is checked against the data type expected by the expression. A value of ARRAY data type cannot be returned by a function as well as a RECORD that contains arrays as members.

Below is an example of a function that returns a whole record. The values returned by the function match the members of the record in number order and data type, so though in the RETURN clause all the variables are listed, the RETURNING clause can contain only the record name with the .* notation:

```
MAIN
DEFINE my_rec RECORD
    var1, var2, INT,
    var4, var5 CHAR (15)
END RECORD
...
CALL my_function() RETURNING my_rec.*
...
END MAIN
FUNCTION my_function()
DEFINE
    r_var1, r_var2 INT,
    r_var4, r_var5 CHAR (15)
...
RETURN r_var1, r_var2, r_var4, r_var5
END FUNCTION
```

The second example is an example of the implicit function invocation for a function that returns only one value:

```
MAIN
DEFINE cl_fullname VARCHAR(25)
...
LET cl_fullname = get_fname(), 1 SPACE, get_lname()
...
END MAIN
FUNCTION get_fname()
DEFINE
```



```
r_name VARCHAR (10)
PROMPT "Enter the first name: " FOR r_fname
RETURN r_fname
END FUNCTION

FUNCTION get_lname()
DEFINE
    r_lname VARCHAR (15)
PROMPT "Enter the last name: " FOR r_lname
RETURN r_lname
END FUNCTION
```

The values of large data types (BYTE or TEXT) cannot be returned by a function, because the returned variables should be referenced by values and the variable of large data types can only be passed by reference. If some changes have been made in values of large data types within a function, they will be visible in the calling routine without being returned by the function.

A returned value of character data type cannot be longer than 511 bytes. No more than 10 lines 511 bytes each can be returned by a function. The actual size of the returned character strings is limited to 512 bytes, but they require the terminating symbols (ASCII 0) at the ends. To pass longer values by reference, use the variables of TEXT data type (see the Passing Arguments section of the CALL statement description).



RUN

The RUN statement is used to specify a command line for the operating system to execute.



Element	Description
Command	A quoted character string or a variable of character data type
Display Mode	The display mode: either IN LINE or IN FORM
RETURNING Clause	A clause that returns the termination status code of the executed command

The RUN statement executes an operating system command. It can be used to run another 4GL application from the first one. When the command invoked by the RUN statement terminates (that is when the second 4GL program is terminated), 4GL resumes the execution of the parent application that contains the RUN statement.

The RUN statement is followed by the command. The command can be represented by:

- a quoted character string that specifies the name of the program and the pathname if necessary
- a variable of CHAR, VARCHAR or STRING data type (it can also be a record member of character data type or an element of a program array of character data type) that contains the name of the program and the pathname if necessary

The example below executes a command line which is specified in an element of an array:

```
RUN my_arr[i]
```

The RUN statement has the following effects, unless the WITHOUT WAITING keywords are specified:

1. The execution of the parent program is suspended
2. The output from the specified command is displayed
3. The execution of the parent 4GL application is restored, the previous 4GL window is restored

If the RUN statement contains the WITHOUT WAITING clause, only the third action from the above list is executed. The command invoked with the WITHOUT WAITING clause is usually executed without affecting the visual display.

RUN command

This is a standard Informix format of the RUN statement which is used to execute command line commands on the system where the application server is located. If you run the parent program in the character mode, the command is executed on the local system. You can run both 4GL child applications and other commands and applications including Windows processes:

```
RUN "notepad.exe"
```



The GUI Mode

If you use this command in GUI mode for running a child 4GL application, this child application must be located on the system where the corresponding application server is running. The child application must be deployed to this application server. A parent program is deployed to the server automatically, when it is run. However, the child program is not run directly and thus is not deployed automatically. To deploy a program to the GUI server, you can use the **Deploy to Server** option from the context menu of the program in LyciaIDE, or just copy the compiled program to the /progs directory of your application server.

	Note: In the GUI mode, if the child 4GL application is not deployed to the application server, an error will occur when the parent application will attempt to launch it
--	---

```
# this command will run "prog3" 4GL application provided that it is
# deployed to the GUI server or that you use the character mode
RUN "prog3"
```

You can optionally add the WITHOUT WAITING and RETURNING clauses to this RUN statement type.

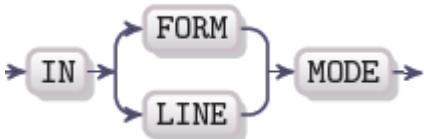
It is not advisable to run another commands which are not 4GL applications using this command in the GUI mode. If a non-4GL application is launched in such a way, it will not communicate sensibly with the thin client. It will execute the command or run the non-4GL application, but it will be run in the context of a non-interactive desktop, hence, it is not visible in the foreground. In addition this will block the execution of the parent program until such child process is terminated, unless the WITHOUT WAITING clause is specified.

Thus it is preferable to use winexec() and winshellexec() functions instead to invoke Windows processes in the GUI mode:

```
CALL winshellexec("notepad.exe")
```

The Default Screen Display Modes

There are two screen display modes, they have the following syntax:



These modes can be specified explicitly in the following statements: OPTIONS, RUN, REPORT, and START REPORT. Different screen display modes can be specified for the RUN statement and for the REPORT statement that sends the output to the pipe by means of the OPTIONS statement.

The IN LINE MODE is the default behaviour for the RUN statement. If no screen display mode is specified, the current value specified in the OPTIONS statement is used.

If the IN LINE MODE is activated, the terminal is in the same state as when the program began. The interrupts are enabled. The input does not become available until a NEWLINE character is typed. If the IN



FORM MODE is activated the terminal input is performed the in raw mode, each character is available for input the moment it is typed.

The IN LINE MODE is the default screen display mode. However, many 4GL statements take it into IN FORM MODE (these are DISPLAY AT, OPEN WINDOW, DISPLAY FORM statements, OPTIONS statement that specifies keys, etc.). That is why many of the 4GL programs are in the IN FORM MODE most of the time.

The IN FORM MODE is the default mode for PIPE. The default screen display mode for RUN is IN LINE MODE. If you try to specify RUN IN FORM MODE in the OPTIONS statement, the program will not enter the formatted mode. However, if it is already in formatted mode, it will remain formatted. The RUN IN LINE MODE switches the program to the line mode, if it has been in formatted mode.

The RETURNING Clause

The RETURNING clause has the following syntax:



Element	Description
Value	An integer or small integer, or a variable of INT or SMALLINT data type

If a RUN statement has the RETURNING clause, the termination status code of the executed command is stored in the variable of INTEGER data type that is specified in the RETURNING clause. This allows 4GL to undertake actions depending on the value of this variable. The "0" status code specifies that the command has been terminated normally. Non-zero values of the variable usually specify a number of errors which caused the execution to terminate.

The RETURNING clause can be used in the RUN statement only if it invokes a child 4GL program which contains the EXIT PROGRAM statement. After such program has been executed, the variable will contain two bytes of the information about the termination status of the program:

- The low byte specifies the termination status of the program executed with the help of the RUN statement. The value of this status can be calculated by using the formula: *integer value* modulo 256.
- The high byte contains the low byte from the EXIT PROGRAM statement of the program executed by the RUN statement. Its value can be recovered by dividing the *integer value* by 256.

The example below consists of a simple 4GL child program and the parent program. The child program is:

```
MAIN
DEFINE r_var INT
LET r_var = 10
EXIT PROGRAM (r_var)
END MAIN
```

Below, the parent program invokes the child program:



```
MAIN
DEFINE err_code, term_code, r_val INT,
      my_program CHAR(15)
...
RUN my_program RETURNING r_val
LET term_code = (r_val MOD 256)
IF term_code <> 0 THEN
  MESSAGE "Unable to run the ", my_program, " program."
END IF
LET err_code = (r_val/256)
DISPLAY my_program, " program returns ", err_code, " error code."
```

If the child program is not terminated before 4GL encounters the EXIT PROGRAM statement, the displayed value of the error code (variable *err_code*) will be 10. You should take cautions calculating the actual error number, because sometimes the (variable)/256 formula may not return the actual error number.

The integer value 256 is assigned to the variable in the RETURNING clause, if a child program is terminated by means of pressing the Interrupt key. Pressing the Quit key will return the integer value (3*256), or 758.

Is the child program can be terminated by actions of the user. Include several EXIT PROGRAM statements with different numbers in different parts of the program. The examination of the value of the variable in the RETURNING clause may give information as to at which point the program has been terminated. This clause cannot be used, if the RUN statement contains the CLIENT SESSION keywords.

The WITHOUT WAITING Keywords

The WITHOUT WAITING keywords are used within the RUN statement to make it execute the child program in the background, without displaying the process of its execution.

```
DEFINE my_prog STRING
LET my_prog = "/child_progs/source/test_prog.4a"
RUN my_prog WITHOUT WAITING
```

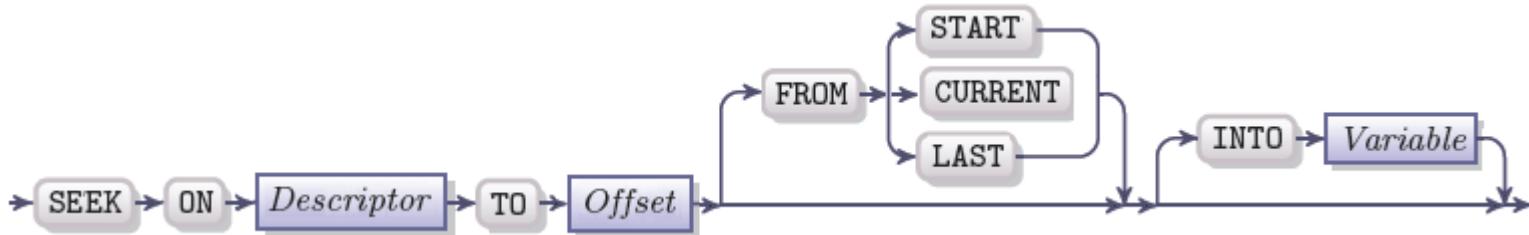
A child application must contain its own MAIN program block. The variables of the two programs cannot be cross-referenced. Each of the programs (both parent and child programs) must be terminated independently by means of the EXIT PROGRAM or END MAIN statement.

The WITHOUT WAITING keywords are usually used when the child program needs some time for execution and the result of its execution is not required for the processing of the parent program. If these keywords are included into the RUN statement, the child program is executed as a background process and does not affect the visual display. This feature is typically used to run 4GL reports in the background.



SEEK

The SEEK statement is used to define the offset in the file starting from which the READ and WRITE statements should read from or write to it. It is usable with the file opened with the OPEN FILE statement.



Element	Description
Descriptor	An integer variable that stores an open file descriptor
Offset	It is an integer variable or integer literal that defines the value for repositioning the offset in the file.
Variable	An integer variable that stores the file offset that results from the SEEK execution.

The START, LAST and CURRENT keywords specify the position from which the offset value is calculated. The default position value is START.

You may want to find out the size of the file before specifying the offset. To do it use the FROM LAST position and store the resulting offset into a variable. E.g.:

```
SEEK ON my_f TO 0 FROM LAST INTO file_size
```

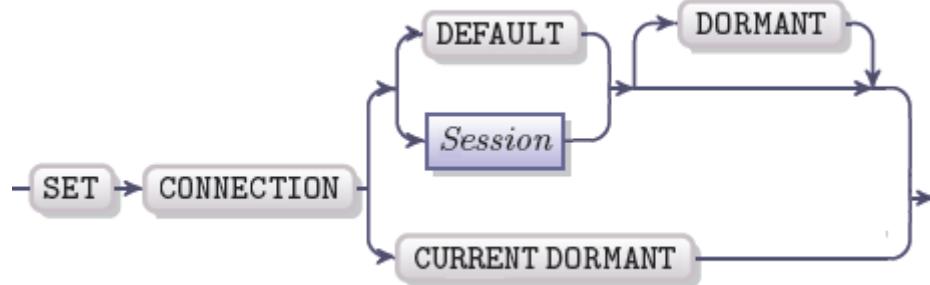
The following example allows you to start reading the data from the file from position20 instead of 0 which is the default offset:

```
OPEN FILE my_f FROM "clients.csv" OPTIONS (READ, FORMAT="csv")
SEEK ON my_f TO 20
READ FROM my_f INTO var1, var2
```



SET CONNECTION

The SET CONNECTION statement changes the current database session. Here is its syntax:



Element	Description
Session	A string expression identifying the session name of the database connection that is to become the current session.

A connection session becomes current automatically right after the corresponding [CONNECT TO](#) statement is executed. If another CONNECT TO is executed later, the session ceases to be the current one. Even if the next current connection is disconnected, the previously opened sessions do not become the current ones unless you explicitly instruct so by using the SET CONNECTION method.

The SET CONNECTION statement cannot be used for the sessions that were disconnected using the [DISCONNECT](#) statement.

SET CONNECTION DEFAULT

The DEFAULT keyword used with the statement sets the connection specified by the DATABASE statement as the current connection. It is also used to set the default connection declared as CONNECT TO DEFAULT as the current connection.

```
DATABASE my_db
MAIN
...
CONNECT TO "ifx_db" -- this is the current connection at this point
...
SET CONNECTION DEFAULT -- the connection to my_db is current
```

SET CONNECTION Session

You can set the current connection to a named database connection session declared previously by the CONNECT TO statement.

```
CONNECT TO "database1" -- current session database1
CONNECT TO "database2" -- current session database2
CONNECT TO "database3" -- current session database3
DISCONNECT CURRENT -- no current session. database3 session was deleted
```



```
...
SET CONNECTION "database2" -- current session database2
```

DORMANT and CURRENT DORMANT Keywords

These keywords are applicable only for connecting to IBM Informix databases.

Adding the DORMANT keyword at the end of the statement makes the specified connection dormant instead of active. It allows you to set a database session inactive instead of disconnecting it. The main difference between setting a session dormant and using the DISCONNECT statement is that a dormant connection can be made active again using the SET CONNECTION statement. This would not be possible, if the session were disconnected.

You can set the current connection as dormant using the CURRENT DORMANT keywords or by specifying its name. However, once the current connection was set dormant, there will be no active connection until another SET CONNECTION or CONNECT TO statement is executed.

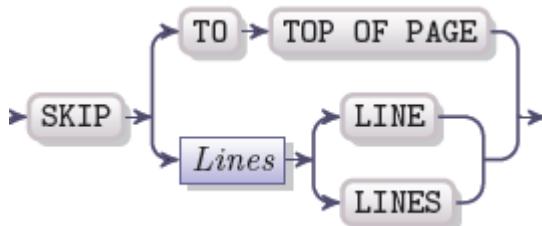
```
CONNECT TO "database1" -- current session database1
CONNECT TO "database2" -- current session database2
CONNECT TO "database3" -- current session database3

SET CONNECTION "database3" DORMANT -- no current session
#beginning from here no SQL statements can be executed
...
SET CONNECTION "database3" -- current session database3
# starting from here SQL statements can be used again
```



SKIP

The SKIP statement can be used only in the FORMAT section of the REPORT program block. It inserts the specified number of empty lines into a report, or skips the remainder of the current report page and starts the new one.



Element	Description
Lines	An integer expression that specifies the number of empty lines to be inserted

The SKIP statement is used to advance the current print position to the next page of the report or to move it down by a specified number of lines. The SKIP keyword can be followed by:

- The TO TOP OF PAGE keywords – the current print position is moved to the top of the next page
- An *integer* LINE structure, which specifies the number of lines by which the current printing position should be moved downwards. The LINES and LINE keywords are interchangeable in this context.

When the printing position is moved, the output produced by the PAGE HEADER and PAGE TRAILER clauses remains in its usual position.

The SKIP TO TOP OF PAGE statement cannot be used in the PAGE HEADER, PAGE TRAILER and FIRST PAGE HEADER clauses. The SKIP statement cannot appear in loops (FOR and WHILE) and in the CASE statement.

```
...
PAGE HEADER
PRINT COLUMN 35, "ORDER INFO"
SKIP 2 LINE
PRINT "ORDER ID", COLUMN 10, "ORDERED BY", COLUMN 30, "ITEMS",
COLUMN 50, "QUANTITY", COLUMN 58, "PRISE", COLUMN 70, "SUM TOTAL"
...
```



SLEEP

The SLEEP statement is used to suspend the execution of a program for a certain period of time which is calculated in seconds.



Element	Description
Integer Expression	A 4GL expression that returns a positive integer

The SLEEP keywords are followed by an integer expression which specifies the number of seconds by which the program execution is suspended. This statement might prove useful, if you want a screen display to remain visible long enough for the user to be able to read it.

The example below illustrates the usage of the SLEEP statement for creation of a pause between two messages. Without the SLEEP statement, the second MESSAGE statement would have erased the first one before it could be read:

```
MESSAGE "You have deleted a row."  
SLEEP 2  
MESSAGE " "
```

The PROMPT statement could be used instead of the SLEEP statement in some cases. The above message can be displayed using the PROMPT statement, the user input will erase the PROMPT message and the program execution will be resumed:

```
PROMPT "The row is deleted. Press any key to continue." FOR CHAR anykey
```

The PROMPT message remains visible until the user presses any key. The value of the *anykey* variable can be ignored.



SQL

The SQL statement has the SQL...END SQL structure, it is used for execution of SQL statements, which cannot be embedded into the 4GL source code. Only preparable statements can be used in this statement, with some exceptions, which are discussed later in this chapter.



Element	Description
SQL Block	A preparable SQL statement that the database server supports

Some of the SQL statement can be embedded into the Querix 4GL (these are almost all of the statements supported by the Informix database version 4.10 and some of the statements that are supported by the later versions). Other SQL statements must be prepared by means of the PREPARE statement, if the database can execute the prepared statements.

The SQL...END SQL statement provides an alternative to the PREPARE statement. It automatically prepares, executes and frees the SQL statements. The example below illustrates the usage of the ALTER TABLE statement with DISABLED keyword. A statement like this is simply put within the SQL...END SQL statement, if it does not produce input and does not require input.

```
SQL
ALTER TABLE order_info MODIFY (client CHAR(20), contact_info DISABLED)
END SQL
```

If you try to embed this statement into the 4GL code, it will produce an error, because DISABLED keyword is not supported by the embedded ALTER TABLE statement.

You can include only SQL statements into the SQL...END SQL statement.

Host Variables

The SQL...END SQL statement can accept host variables, which should be prefixed by a dollar sign (\$) either with or without a white space between the sign and the name of the variable. The host variables serve as input and output parameters.

```
SQL
INSERT INTO my_table (first_col, second_col)
SELECT TRIM(A.fname) || " " || TRIM(A.lname),
B.whatever FROM my_table2 A, my_table3 B
WHERE A.Filter = $my_array[i].item
AND B.Filter MATCHES $var1
```



```
END SQL
```

The variables prefixed by the dollar symbols are not the database entries, they are host variables. Sometimes SQL identifiers are prefixed by the “at” symbol (@) to distinguish them from the 4GL identifiers.

Returned Values

The values obtained by the SQL...END SQL statement can be returned to the 4GL program, if you use the SELECT INTO and EXECUTE PROCEDURE INTO SQL statements. The host variables and returned values stick to the same rules (they must be prefixed by the dollar sign):

```
SQL
SELECT someProced ($var2)
INTO $f_var, FROM someTabl WHERE PkColumn=$pkval
END SQL
```

The SELECT INTO *variable* and EXECUTE PROCEDURE INTO SQL statements can be used within the SQL...END SQL statement but they cannot be prepared by means of the PREPARE statement. This is an exception to the rule that only those SQL statements that can be prepared can be used in the SQL...END SQL statement.

Cursors in SQL...END SQL Statement

The cursor names are not mangled within the SQL...END SQL statement. You must ensure that all the conflicts connected with the cursor name are resolved, before you can reference a cursor within the SQL...END SQL statement. The declaration of a cursor must precede the SQL...END SQL statement in which it is referenced.

```
DECLARE c_su SCROLL CURSOR WITH HOLD FOR
SQL
SELECT TRIM(Firstname)||" "|| TRIM (Lastname)
INTO $var1 FROM someTable WHERE PkColumn > $pkvar
END SQL
```

Statements Excluded from the SQL Block

Only preparable statements can be used in the SQL...END SQL statement, with some exceptions. The SQL...END SQL block can include only SQL and SPL statements (not 4GL statements). Below is the list of some SQL statements that are used in some Informix versions, but which are not supported by the SQL...END SQL statement:

ALLOCATE COLLECTION	FREE
ALLOCATE DESCRIPTOR	GET DESCRIPTOR
ALLOCATE ROW	GET DIAGNOSTICS
CHECK TABLE	INFO
CLOSE	LOAD
CONNECT	OPEN
CREATE PROCEDURE FROM	OUTPUT



DEALLOCATE COLLECTION	PREPARE
DEALLOCATE DESCRIPTOR	PUT
DEALLOCATE ROW	REPAIR TABLE
DECLARE	SET AUTOFREE
DESCRIBE	SET CONNECTION
DISCONNECT	SET DEFERRED_PREPARE
EXECUTE	SET DESCRIPTOR
EXECUTE IMMEDIATE	UNLOAD
FETCH	WHENEVER
FLUSH	

You cannot use the CREATE PROCEDURE statement in the SQL...END SQL block. Use the embedded CREATE PROCEDURE FROM *file-name* statement instead.

Restrictions

- The PREPARE and SQL...END SQL statements cannot be included into one another.
- The question marks (?) that replace the user supplied values can be used only in the prepared statements, **not** in the SQL...END SQL statements.
- If you include a semicolon symbol (;) after an SQL...END SQL statement, it will produce no error but will also have no effect.
- If you separate two statements within the SQL...END SQL statement with the help of the semicolon (;), a compile-time error will occur.

Comment Symbols and Optimization

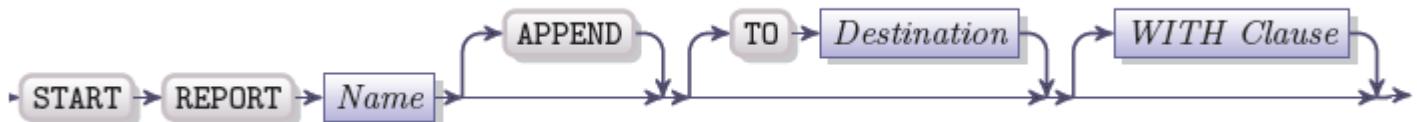
The hash symbol (#) cannot be used as a comment symbol within the SQL...END SQL block. Only the double braces ({ }) and double hyphen (--) symbol to mark the comments.

The optimization directives are passed to the database symbols, if they are created by means of the standard notation for these features (in Informix version 7.3 and above). The optimization directives should be enclosed in braces and preceded by the plus symbol (+) which indicates that the code within the braces is an optimizer directive. The directives can follow the DELETE, SELECT, or UPDATE statements within the SQL...END SQL statement.



START REPORT

The START REPORT statement is used in the report driver to indicate the beginning of the report processing. It can contain information about the dimensions and destination of the report output.



Element	Description
Name	The name of a report declared in the corresponding REPORT program block
Destination	The destination of the report output
WITH Clause	Optional clause which specifies the dimensions of the report page

When 4GL encounters the START REPORT statement, it performs the following actions:

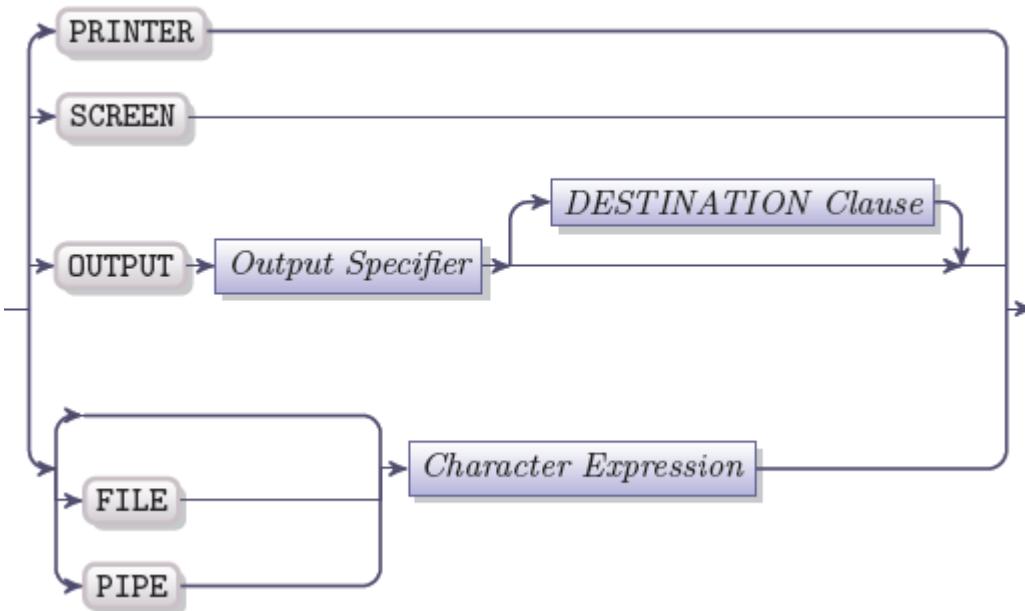
- Identifies the REPORT program block that corresponds to the report identifier used in the statement. The REPORT program block contains the criteria for formatting the report
- Specifies the dimensions of a page and destination for the output
- Initializes page headers that are specified in the REPORT program block

If you are satisfied with the default delimiters and destination, you need to specify only the report name in the START REPORT statement. The output settings of the START REPORT statement override the settings of the OUTPUT clause of the REPORT program block.

If you use the START REPORT statement with the identifier of a report that is already running, you may receive an unexpected result. The START REPORT statement is usually followed by a loop (FOR, FOREACH or WHILE). It contains the OUTPUT TO REPORT statement which sends the data retrieved by each iteration of the loop to the report. The loop should be followed by the FINISH REPORT statement which terminates the processing of the report.

Report Destination

The TO clause is optional, it specifies the destination for the output. If this clause is present in the START REPORT statement, it overrides the settings of the REPORT TO clause of the corresponding REPORT program block.



Element	Description
Character Expression	A 4GL character expression that returns a valid destination
Output Specifier	A character string or a variable of character data type
DESTINATION Clause	A clause that specifies a file name or a program name

The TO clause of the START REPORT statement overrides the settings specified in the REPORT TO clause of the REPORT program block. If the TO clause is omitted, the output destination for the report is the destination specified in the REPORT program block, in the REPORT TO clause. If there is no TO clause in the START REPORT and no REPORT TO clause in the REPORT program block, the output is made to the screen. Specifying the SCREEN keyword in the TO clause has the same effect.

If an empty set of data is sent to the report by the OUTPUT TO REPORT clause, or if there is not OUTPUT TO report clause, no output is produced even if you specify the START REPORT CLAUSE, and the TO clause as well as the REPORT TO clause has no effect.

The TO clause can send the output to the following destinations:

- To the screen (if the SCREEN keyword is specified)
- To a printer (if the PRINTER keyword is specified)
- To a file (if the FILE keyword is specified)
- To the command line, another program or shell script (if the PIPE keyword is specified)

Output Configurations

You can optionally use the TO clause to specify the output destination. The REPORT TO clause of the REPORT program block specifies the permanent destination for the output. The TO clause can specify another destination for the output at runtime.



- In the TO FILE file name or in the TO PIPE program the program and file name can be program variables, to which different values can be assigned at runtime
- The TO OUTPUT clause can specify the SCREEN, PRINTER, FILE, or PIPE keyword as the destination

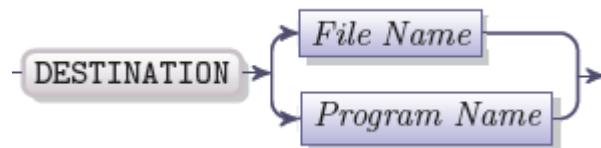
The program and the file name should be specified as quoted character strings, if they are not represented by a variable that contains a character string.

The OUTPUT... DESTINATION Keywords

The OUTPUT keyword must be followed by the output specifier. The specifier can be either a quoted character string or a variable of character data type. Here are the valid output specifiers:

- "SCREEN"
- "PRINTER"
- "FILE"
- "PIPE"
- "PIPE IN FORM MODE"
- "PIPE IN LINE MODE"
- A character variable which returns any of the above listed character strings

The DESTINATION clause is optional and should be included into the TO OUTPUT clause only if the output is made to a file or to a pipe. It has the following syntax:



Element	Description
File Name	A variable or a quoted character string returning a valid file name
Program Name	A variable or a quoted character string returning a valid program name

If the PRINTER or SCREEN keyword is specified as the output specifier, the DESTINATION keyword is not required and is ignored, if included.

If "FILE", "PIPE", "PIPE IN FORM MODE", "PIPE IN LINE MODE" or a variable returning these strings is specified as the output specifier, the DESTINATION keyword. The example below uses the TO OUTPUT clause to perform the output to a file:

```
DEFINE rep_file, rep_destin STRING
LET rep_file = "file0045.txt"
LET rep_destin = "file"
START REPORT my_rep1 TO OUTPUT rep_detsin DESTINATION rep_file
```

The usage of the TO OUTPUT ... DESTINATION clause for each report destination is discussed in detail below in the corresponding sections.



The SCREEN Keyword

If the TO clause is followed by the SCREEN option, the output is sent to the Report window.

```
START REPORT ord_rep TO SCREEN
```

The same effect may be achieved by using the DESTINATION clause:

```
START REPORT ord_rep TO OUTPUT SCREEN
```

If you use the output to screen, you may want to define the number of lines contained by each page of the report not more than the total number of lines on the screen of your display. It can be done by means of the PAGE LENGTH keywords, a page will be no longer than the value specified together with these keywords.

To ensure that the report remains on the screen long enough for a user to read it, you should include the PAUSE statements in the FORMAT section of a report, if the application is run on Linux/UNIX OS.

The PRINTER Keyword

If the TO clause is followed by the PRINTER option, the output is made to the printer, specified by the DBPRINT environment variable. If the environment variable is not set, the output is sent to the default printer specified in the operation system.

```
START REPORT ord_rep TO PRINTER
```

The same effect may be achieved by using the DESTINATION clause:

```
START REPORT ord_rep TO OUTPUT "PRINTER"
```

You can send the output to non-default printers, to do it use one of the following ways:

- Set the desired printer in the DBPRINT environment variable and then use the TO PRINTER clause or the TO OUTPUT "PRINTER" clause
- Use the START REPORT with the TO FILE "*file-name*" clause or TO OUTPUT "FILE" DESTINATION "*file-name*" clause to send the output to a file and then print that file
- Use the START REPORT with the TO PIPE "*program*" clause or TO OUTPUT "PIPE" DESTINATION "*program*" clause to send the output to a command line or to a shell script that redirect it to the printer

The FILE Keyword

If the TO clause is followed by the FILE option, the output is made to the specified *file-name*. The FILE keyword is optional, it is required only if you want to send output to a file using the DESTINATION clause:

```
START REPORT ord_rep TO OUTPUT "FILE" DESTINATION "ord_rep_file"
```

In other cases only the *file-name* can be used:

```
START REPORT ord_rep TO "ord_rep_file"
```



A *file-name* can be either a character string enclosed into quotation marks or a variable of character data type that stores such string.

The PIPE Keyword

If the TO clause is followed by the PIPE option, the output is made to another program, command line or shell script. It can override either the IN LINE MODE or IN FORM MODE settings specified in the OPTIONS statement for screen output. The PIPE option should be followed by:

- command line arguments
- a name of a program enclosed in quotation marks
- a variable of character data type.

Unlike the FILE keyword, the PIPE keyword is always required when you make output to a pipe:

```
START REPORT ord_rep TO PIPE "/output/rep/destp"  
START REPORT ord_rep TO OUTPUT "PIPE" DESTINATION "/output/rep/destp"
```

The START REPORT statement can specify whether the output should be in the IN LINE MODE or IN FORM MODE:

```
START REPORT ord_rep TO PIPE "/output/rep/destp" IN LINE MODE  
START REPORT ord_rep TO OUTPUT "PIPE IN FORM MODE" DESTINATION  
"/output/rep/destp"
```

If the screen display mode is not specified in the TO PIPE clause, the settings of the OPTIONS statement will be used.

The APPEND Keyword

The APPEND keyword can be added before the TO clause, if you want to append your report to a previously generated report. This feature is available only if both reports – the new and the old – are output to a file. Appending works only when the output is sent to a file, because reports sent to a printer are always appended and it is the natural course of things and reports sent to the screen are not overwritten or appended while they are opened in different instances of the Report Viewer.

If you omit the APPEND keyword and send a report to a file which contains a previously produced report, the contents of the file will be overwritten by the new report and the data will be lost. If the APPEND keyword is present, the new report will be added to the existing one and appended to the end of the specified report file.

```
START REPORT rep2 APPEND TO FILE "C:\\f_rep.txt"
```

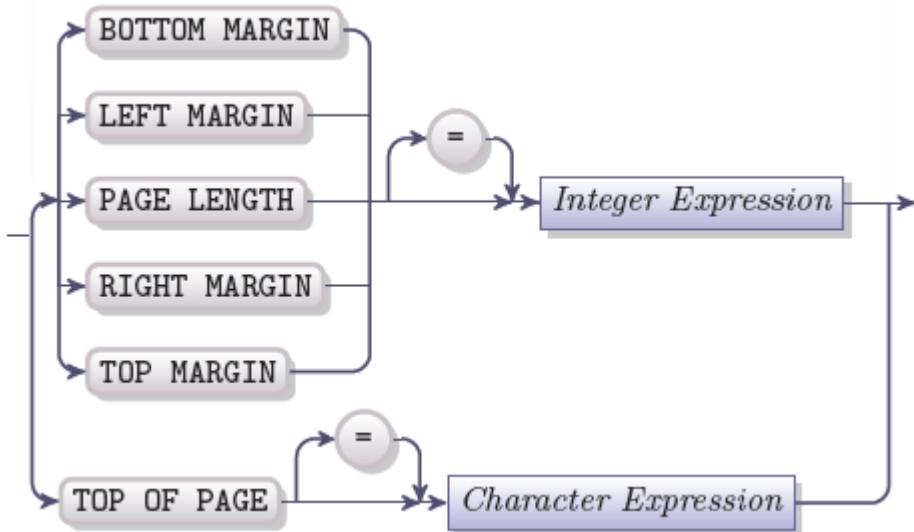
Using the APPEND keyword you can append reports unlimited number of times and no information will be lost.

The WITH Clause

The WITH clause sets the size of the report pages. It overrides the dimension settings in the OUTPUT clause of the REPORT program block. The WITH keyword can be followed by the number of lines that determine the

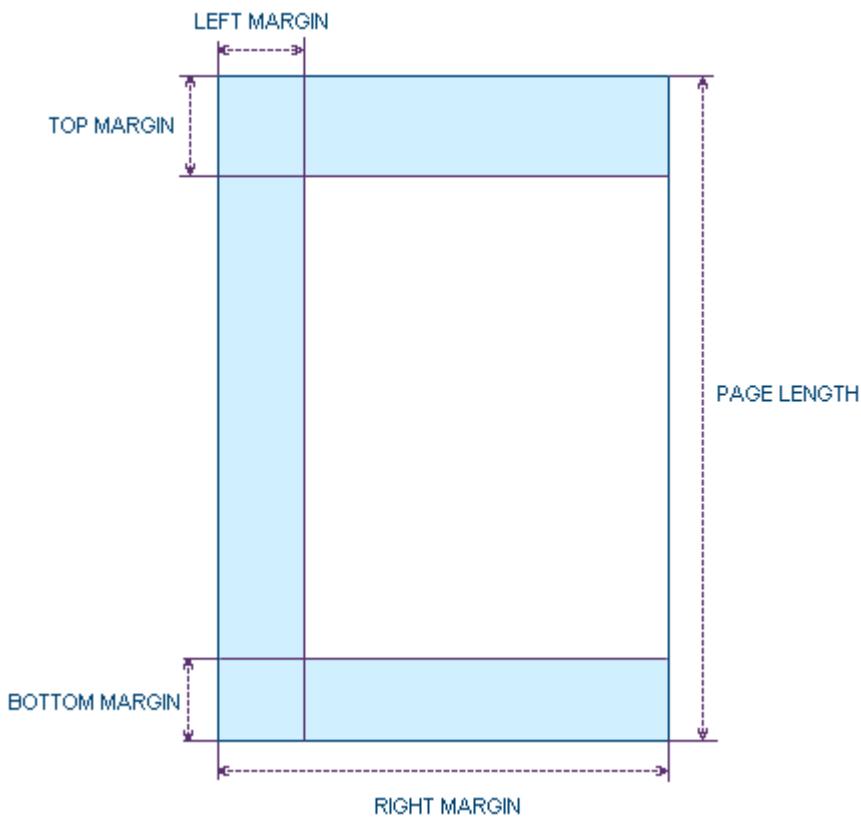


page height and the number of characters that determine the page width and with other data that determine margins. The syntax of the WITH clause is as follows:



Element	Description
Integer Expression	An expression returning a positive integer or a literal integer no larger than 2,766.
Character Expression	An expression that returns a quoted character string that specifies the page-eject character sequence

Below is the diagram of a report page, with the positions of all the possible delimiters:



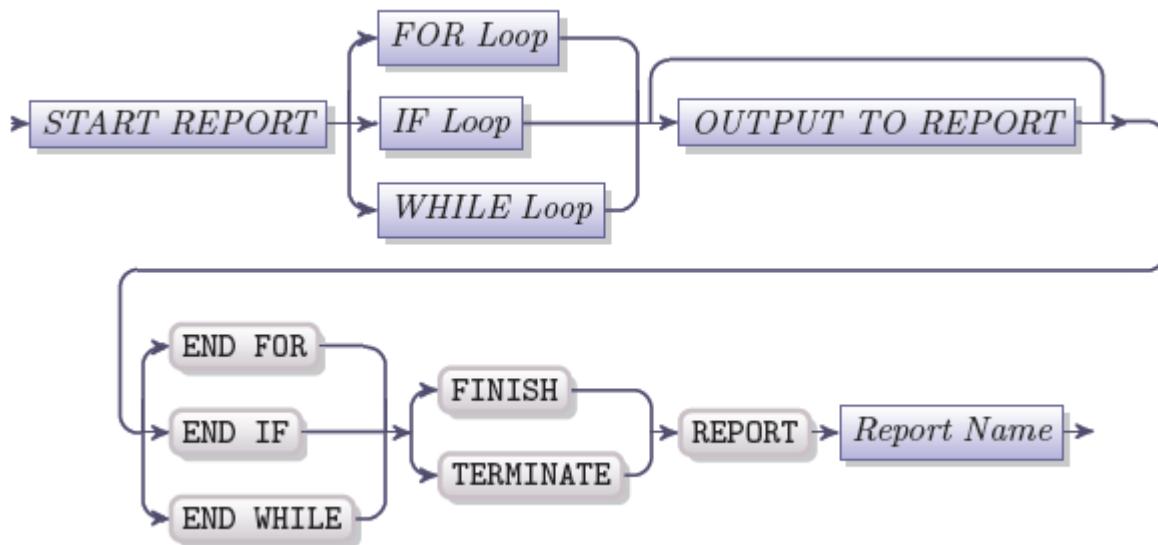
- The RIGHT MARGIN keywords specify the page width together with the left margin. It can be omitted, if you specify FORMAT EVERY ROW option, then the default value will be used, which is 132 characters
- The LEFT MARGIN keywords specify the number of blank spaces with which each line of the report output should start. By default it is 5 characters
- The PAGE LENGTH keywords specify the page length together with the top and bottom margins as well as the page headers and trailers. By default it is 66 lines
- The TOP MARGIN keywords specify the number of empty lines that appear at the beginning of each report page above the first line of the actual output text
- The BOTTOM MARGIN keywords specify the number of empty lines that follow the last line of the actual output text at the bottom of each report page

You can also use the TOP OF PAGE keywords to specify a page-eject sequence for a printer. If you specify this value, it may reduce the time needed for producing a large report.

For more detailed information about the report page attributes, see the chapter on the [OUTPUT report section](#).

The Report Driver

The START REPORT statement is the beginning of a report driver which is placed outside the REPORT program block and is used to invoke a 4GL report. The report driver invokes a report, and sends data to it.



Element	Description
START REPORT	The START REPORT statement with all the optional clauses if necessary
FOR Loop	The FOR statement with all its clauses except for the END FOR keywords
IF Loop	The IF statement with all its clauses except for the END IF keywords
WHILE Loop	The WHILE statement with all its clauses except for the END WHILE keywords
OUTPUT TO REPORT	The OUTPUT TO REPORT statement
Report Name	The name of the report which has been used in the START REPORT statement of this driver

A typical report driver consists of:

1. The START REPORT statement which invokes the report processing.
2. A loop created by means of the FOR, FOREACH or WHILE statement which retrieves information from a database.
3. The OUTPUT TO REPORT statement which sends the retrieved information to the report. There can be more than one OUTPUT TO REPORT statement in one report driver.
4. The FINISH REPORT statement that ends the processing of a report.

A report driver can also contain the TERMINATE REPORT statement, which is used for termination of a report, usually it is executed, if an exception condition is met. It is also possible to use the EXIT REPORT statement in the REPORT program block to terminate the report.



TERMINATE REPORT

The TERMINATE REPORT statement stops the processing of a report. It is not used for finishing report normally (use the FINISH REPORT for this purpose), it is typically executed when an error occurs during the report processing. This statement is only valid in the report driver and not in the REPORT program block. The EXIT REPORT is used in the REPORT program block for the same purpose.



Element	Description
Report Name	The name of a 4GL report, as declared in a REPORT program block

The TERMINATE REPORT statement must be followed by the identifier of a report that is defined in the corresponding REPORT program block. This statement must follow a START REPORT statement that references the same report identifier. When 4GL encounters the TERMINATE REPORT statement, it exits the report driver without completing the report. 4GL terminates the processing of the report and deletes temporary files and tables that were created during the execution of the report.

The TERMINATE REPORT neither formats the values from aggregate functions nor executes the ON LAST ROW statement, if it is specified in the REPORT program block. The TERMINATE REPORT statement make the report driver produce an empty report, if the report includes the ORDER BY clause (unlike the reports with the ORDER EXTERNAL BY option).

The TERMINATE REPORT statement can be used when you do not need a report that misses some records and 4GL detects the situation in which some of the data are prevented from being reported. E.g. something disables the program to which the report is sent through a pipe.

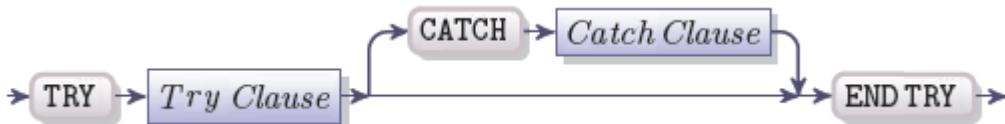
It is advisable that you use the TERMINATE REPORT statement in conditional statements that detect errors which may occur while processing the report. If no errors occur, the report should normally be completed by means of the FINISH REPORT statement. The example below shows the situations in which the FINISH REPORT and TERMINATE REPORT statement are used:

```
IF err_val > 0 THEN
    TERMINATE REPORT cust_1
ELSE
    FINISH REPORT cust_
END IF
```



TRY...CATCH

The TRY...CATCH statement is one of the statements used for error handling. The general syntax of the statement is as follows:



Element	Description
Try Clause	A set of statements that may cause a runtime error
Catch Clause	A set of statements handling the occurred error

When the program encounters the TRY keyword, it begins executing the statements specified in the *TRY Clause*. These can be any 4GL or SQL statements. If any exception occurs during the statements execution, the program control passes to the *CATCH Clause* which contains the statements that are to be executed in case of the error occurrence.

The execution of the source code below must result in a table creation. If the table already exists, an error will occur. In this case, the CATCH block opens a dialog window which allows user to chose whether to use the existing table or delete it.

```

TRY
    CREATE TABLE works_list
    (
        obj_title      VARCHAR(100),
        object         BYTE,
        obj_descr      VARCHAR(200),
        obj_create_date DATE,
        obj_info       VARCHAR(200)
    )
    DISPLAY "Table created" AT 2,2

CATCH
    LET answer = fgl_winquestion("Table already exists",
        "Do you want to delete the existing one?",
        "No", "Yes|No", "Question", 1)
    IF answer = "Yes"
        THEN DROP TABLE works_list
    END IF

```



```
CREATE TABLE works_list
(
    obj_title      VARCHAR(100),
    object         BYTE,
    obj_descr      VARCHAR(200),
    obj_create_date DATE,
    obj_info        VARCHAR(200)
)
DISPLAY "Table created" AT 2,2

END IF

END TRY
```

In the example above, if we still want to use the table after we dropped it, we need to use the CREATE TABLE statement again or use any other program execution control statements to return to the beginning of the TRY statement. Otherwise the table will be dropped, but not created, because the TRY...CATCH statement is not a loop.

If no CATCH clause is specified in the statement, the program control goes to the statement following the END TRY keywords in case of an exception.

If all the statements in the TRY clause are executed without exceptions, the program execution continues from the statement following the END TRY keywords.

It is also possible to nest a TRY statement into another TRY statement, if needed:

```
TRY
    SELECT * INTO myrec.* FROM tab_1 WHERE col1 = 1
CATCH
    TRY
        DATABASE default_db
        SELECT * INTO myrec.* FROM tab_1 WHERE col1 = 1
    CATCH
        ERROR "An error occurred during the SELECT statement execution"
    END TRY
END TRY
```



Note: The TRY statement is a so-called pseudo statement. The compiler does not generate code for it. That's why you cannot set a break point at TRY, CATCH or END TRY keywords in the debugger.

The TRY statement tracks the errors only in statements and operators specified within the TRY block. If the TRY clause invokes a programmer defined function, the errors in that function need special handling. Therefore, if an exception occurs in the function called from the TRY block, the program control won't be passed to the CATCH clause of the calling TRY statement, unless the function body contains the WHENEVER ERROR statement with the RAISE action (see WHENEVER statement description). The TRY clause can be compared with the WHENEVER ANY ERROR GOTO *label* construction. The execution of the extract given below will have the same results as that of the first example given in this section:

```
WHENEVER ANY ERROR GOTO catch_error

CREATE TABLE works_list
(
    obj_title      VARCHAR(100),
    object         BYTE,
    obj_descr      VARCHAR(200),
    obj_create_date DATE,
    obj_info       VARCHAR(200)
)
DISPLAY "Table created" AT 2,2

GOTO no_error
LABEL catch_error:

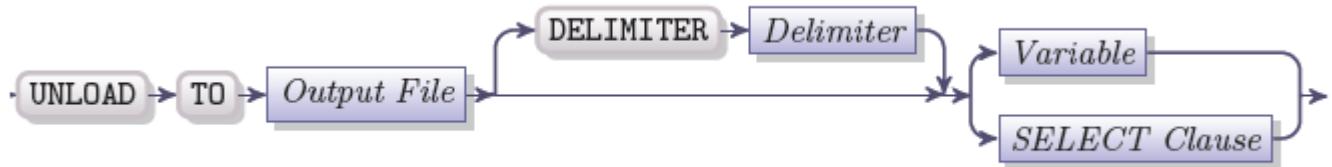
LET answer = fgl_winquestion("Table already exists",
    "Do you want to delete the existing one?",
    "No", "Yes|No", "Question", 1)
IF answer = "Yes"
    THEN DROP TABLE works_list
END IF

LABEL no_error:
...
```



UNLOAD

The UNLOAD statement unloads the data from a table in the current database to an output file.



Element	Description
Output File	A character expression that returns the name of the output file and its relative path if necessary
Delimiter	is a quoted string (or a CHAR, STRING, or VARCHAR variable) containing a delimiter symbol
Variable	A CHAR, STRING or VARCHAR variable containing an SELECT clause.
SELECT Clause	A CHAR, STRING, or VARCHAR variable containing an INSERT clause.

The UNLOAD statement should include a SELECT statement, which specifies the rows in a database that should be copied to a file. The values in the database selected and copied by the UNLOAD statement are not altered or deleted from the table. To execute the UNLOAD statement, the user must have the SELECT privileges on every column from which the values are selected. (The privileges are set by means of the SQL statement GRANT).

The DATABASE or CONNECT statement must precede the UNLOAD statement. You cannot prepare the UNLOAD statement using the PREPARE statement.

Output to a File

The output of the rows selected from a table is performed into an output file, which is specified in a form of a character string enclosed in quotation marks or in the form of a character variable returning such string, the pathname can be also included. The output file can contain only ASCII characters (in default U.S. English locale). In other locales, the output file can consist of the characters that belong to the code set of the locale.

Each row retrieved from the database is sent to the output file in a form of an output record, each of the output records is terminated by the new line character (ASCII 10). Each value within an output record is stored as string of ASCII characters, the data types of the values correspond to the data types of the columns from which the values have been retrieved.

Data Type of a Column	Format of the Retrieved Value
CHAR / VARCHAR / STRING / TEXT	All the trailing blanks are truncated from the values of CHAR and TEXT data types, but they are not deleted from the VARCHAR or STRING values. Each literal delimiter sign, a literal backlash (\) or a newline symbol is preceded by one more backlash symbol in the values of the VARCHAR data type. The backlash symbol is also used as the last



	character in the values of the TEXT data type.
DECIMAL / FLOAT /	The leading blanks are discarded, the values are written as literals.
INTEGER / MONEY /	MONEY values are written without currency symbols. 0 stands for zero
SMALLFLOAT /	values of INT and SMALLINT data types, 0.00 is used for FLOAT,
SMALLINT	SMALLFLOAT, DECIMAL and MONEY zero values. The values retrieved from columns of SERIAL data type are represented by literal integers.
DATE	The values are written in the <i>mm/dd/yyyy</i> format, if other format is not specified in the DBDATE environment variable settings.
DATETIME /	The DATETIME values are written in the format
INTERVAL	<i>yyyy-mm-dd hh:mm:ss.fff</i> with all the time units present or as a continuous subset of the time units. The INTERVAL values are written in the format <i>yyyy-mm</i> or <i>dd hh:mm:ss.fff</i> with all the time units present or as a continuous subset of the time units. Both values are written without the DATETIME and INTERVAL keywords. The time units outside the continuous subset of the time units declared by means of the qualifiers are omitted.
BYTE	The values are written in ASCII hexadecimal format. No whitespaces or newline characters are added. The ASCII hexadecimal form of a BYTE value might be very long and thus difficult to edit or print.

The NULL values of any data type are represented by double delimiter symbol without any blank spaces between the delimiters:

```
12 | John | Smith | | USA
```

The following UNLOAD statement copies values from all the columns of a table which correspond to the condition of a WHERE clause into the *unld_file*:

```
UNLOAD TO "unld_file"
SELECT * FROM ord_inf WHERE ord_date>01/01/2009
```

The content of the *unld_file* will look like the following text:

```
1034|TransCorp|05/10/2009|1500|2.984
1034|PlatoCorp|06/23/2009|200|484
```

The formats of the values that are copied to a file are determined by the settings of the DBFORMAT, DBMONEY, DBDATE, GL_DATE, and GL_DATETIME environment variables.

The DELIMITER Clause

The default delimiter which separates one value from another in an output record is a pipe symbol (|). This default setting can be changed by means of the DELIMITER clause of the UNLOAD statement. The delimiter symbol must terminate an input record only if the last value of this record is NULL.

Below is an example of the UNLOAD statement with the DELIMITER clause which sets the exclamation mark (!) symbol as the delimiter. The UNLOAD statement copies the data contained in all the columns of the *id_num* table into a file. This way the table can be populated by loading this file:



```
UNLOAD TO "/source/unl/unld_info" DELIMITER "!"  
SELECT * FROM id_num
```

If the DELIMITER clause is omitted, the delimiter set by the DBDELIMITER environment variable, if it is set, otherwise the default delimiter symbol pipe (|) is used.

The following symbols cannot serve as delimiters:

- Hexadecimal numbers (0 through 9, a through f, or A through F)
- NEWLINE or CONTROL-J
- Backslash symbol (\)

Host Variables

You cannot substitute host variables with question mark symbols (?) in the SELECT statement, it will lead to the binding problems. Below is the example of the correct usage of the host variables:

```
FUNCTION f_unload()  
DEFINE      filename CHAR(20),  
            delim CHAR(1)  
            i,j,k INTEGER  
  
LET i = 100  
LET j = 30  
LET k = 400  
LET delim = ">"  
LET filename = "/dev/tty"  
UNLOAD TO file DELIMITER delim  
SELECT * FROM my_tabs  
WHERE item_id >= i AND n_cols >= j AND rowsize >= k  
END FUNCTION
```

The Backslash Symbol

The backslash symbol (\) is used as an escape character to indicate that the next character used in the output file should be interpreted as a literal. The backslashes are automatically inserted by the UNLOAD statement to prevent some characters from being interpreted as special characters and not literals.

It is inserted in the following situations:

- If a backslash character is placed in a value of the CHAR, VARCHAR , STRING or TEXT data types
- If a delimiter character is placed in a value of the CHAR, VARCHAR, STRING or TEXT data types (the delimiter character is either the default delimiter - pipe symbol, or the delimiter specified in the DELIMITER clause)
- If a newline character is placed in a value of the VARCHAR data type
- If a newline character is inserted as the last character of a TEXT value



If the LOAD statement is used to insert the values from the file previously created by the UNLOAD statement into a database, all the automatically inserted backlash symbols are deleted.



VALIDATE

The VALIDATE statement tests the values against the range of the values allowed for the corresponding column. To be able to use the VALIDATE statement, you must specify the default database by including the DATABASE statement before any program block within the same module.



Element	Description
Variable List	The list of variables to be validated
Column List	The list of columns against which the values must be validated

This statement is used to validate the user entered data before they will be inserted into a table. The validation criteria can be specified as the attribute of a form field. If the values are inserted into a table not from a screen form field, the VALIDATE statement can apply the criteria for validation.

The values cannot be validated against the columns of large data types or of structured data types. However, you can validate a value against a member of a record or an element of an array that is of simple data type.

The Variable List

The variable list consists of one or more variables separated by commas. The variables can be:

- Variables of simple data types
- Record members including sequences *record.first THRU record.last*. and *record.**
- Array members, but not the complete arrays

The LIKE Clause

The LIKE clause specifies the table columns against which the data should be validated. One VALIDATE statement can contain several variables which are tested, thus the LIKE clause can contain several database columns, against which the values are tested. The columns must match the variables in number, order and they must be of compatible data types. The names of the columns must be prefixed by the names of the corresponding tables.

```
VALIDATE var1, var2, var3 LIKE table1.column4, table3.column2,  
table12.column7
```

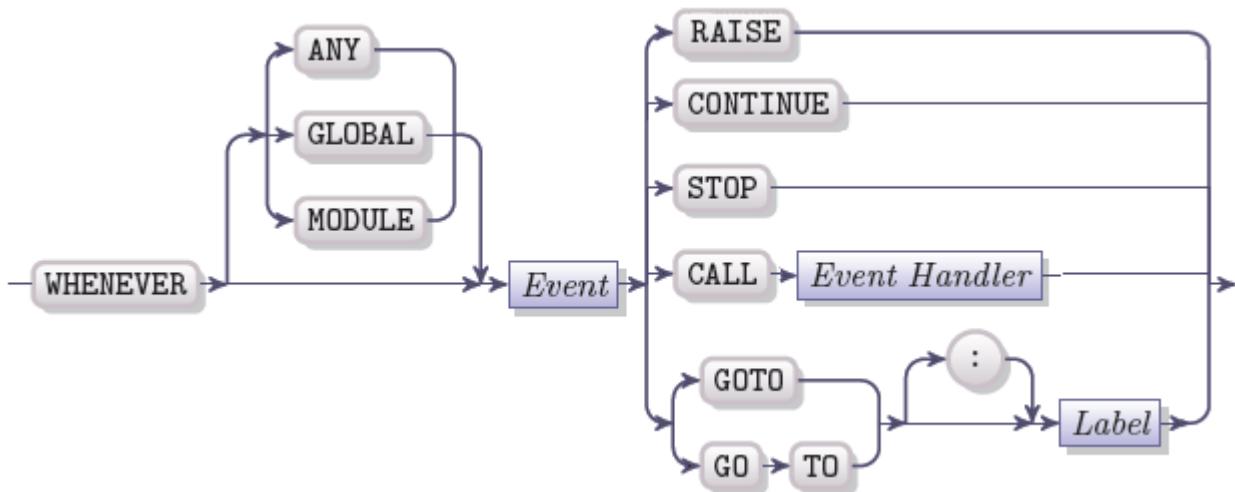
In an ANSI-compliant database, each table name must be preceded by the owner qualifier. The owner qualifier can be omitted only for the tables the owner of which executes this VALIDATE statement.

To use the columns from tables which do not belong to the default database in the LIKE statement, use the qualifier of the server to prefix the table qualifier.



WHENEVER

The WHENEVER statement is used to specify an action that is to be performed, when 4GL encounters an error, a warning, an end-of-data condition or other event during the execution of a program.



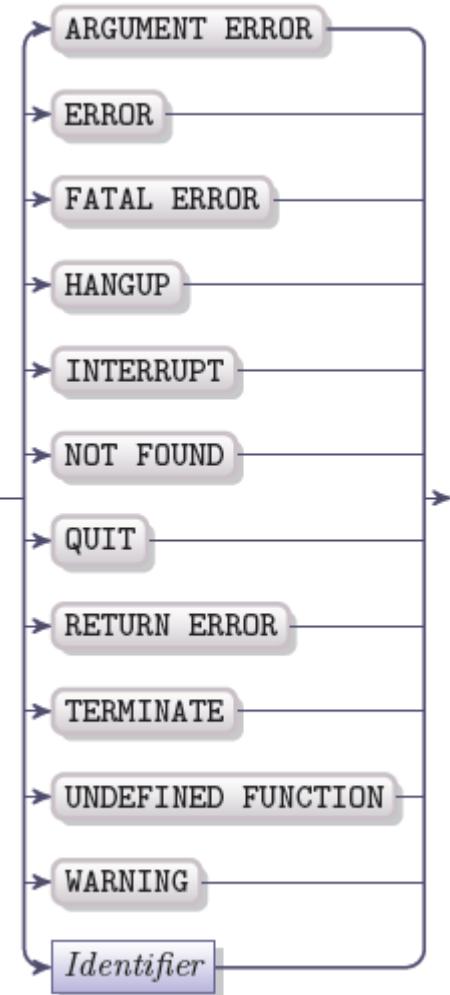
Element	Description
Event	An exceptional condition which can be trapped using the WHENEVER statement
Event Handler	A programmer-defined or built-in 4GL function with no parentheses and no argument list
Label	A previously declared label

The WHENEVER statement cannot appear outside a program block, it must occur either in the MAIN, FUNCTION or REPORT program block.

The WHENEVER statement must include the identification of an exceptional condition, and the identifier of the way in which 4GL should react to this condition if it occurs during the execution of a program.

Trapped Events

Querix 4GL supports some additional events which can be trapped using the WHENEVER statement in addition to the ERROR, WARNING, and NOT FOUND. Here is the complete list of events available in Querix 4GL:



Element	Description
Identifier	A character variable that returns a valid event name

The events are triggered under the following conditions:

Event	Condition of Occurrence
ARGUMENT ERROR	When incorrect number or type of variables are passed to a function call.
RETURN ERROR	When incorrect parameters are returned from a function.
UNDEFINED FUNCTION	When a called function is not defined.
INTERRUPT	When the Interrupt key is pressed.
QUIT	When the Quit key is pressed.
TERMINATE	When the process is being terminated.
HANGUP	When the connection to the client or terminal has closed.
FATAL ERROR	When a fatal application error has occurred (i.e. segmentation fault).



All of these events can be trapped in the same way as a traditional WHENEVER ERROR statement, although typically most usefully they will be defined at the global level, e.g.:

```
WHENEVER GLOBAL QUIT CONTINUE
```

The ERROR Event

When 4GL encounters the WHENEVER statement followed by the ERROR keyword, it takes the specified action, if the value of the sqlcode member of the sqlca built-in variable of the RECORD data type becomes negative after any SQL statement. The same action is taken, if a VALIDATE statement or a screen interaction statement fails. The example below forces 4GL to ignore error and continue execution of the program:

```
WHENEVER ERROR CONTINUE
```

In this context SQLERROR keyword is a synonym for ERROR keyword.

The ANY ERROR Event

The WHENEVER ANY ERROR statement resets the value of the sqlcode member of the sqlca built-in variable of the RECORD data type not only when a VALIDATE statement, a screen interaction statement or an SQL statement fails, but also when any 4GL statement fails.

The –anyerr command line option can override the WHENEVER settings in determining whether the status variable should be reset or not.

The NOT FOUND Event

If the WHENEVER NOT FOUND condition is in effect, the SELECT and FETCH statements are treated differently as compared to other SQL statements. The same can be said about the FETCH and SELECT statements that are implied by the FOREACH and UNLOAD statements.

The action specified in the WHENEVER NOT FOUND statement is undertaken when:

- A FETCH statement tries to retrieve a row from outside the active set
- A SELECT statement returns no rows

The sqlcode built-in variable is set to 100 in both cases.

The NOT FOUND statement has the same meaning as the NOTFOUND statement but these keywords are not interchangeable. Only the NOT FOUND statement can be used in the WHENEVER statement.

The WARNING Event

The WHENEVER WARNING statement (or WHENEVER SQLWARNING statement, which is a synonym) triggers the specified action after any SQL statement which generates a warning. When a warning occurs, the first field of the sqlawarn built-in variable is set to the value "W". If you use the following statement in your program, the execution of a program is halted when a warning occurs:

```
WHENEVER WARNING STOP
```



The ARGUMENT ERROR Event

If the WHENEVER ARGUMENT ERROR statement is in effect, 4GL takes the specified action when it encounters that the incorrect number or type of variables are passed to a function call.

```
WHENEVER ARGUMENT ERROR CONTINUE
```

The RETURN ERROR Event

If the WHENEVER RETURN ERROR statement is in effect, 4GL takes the specified action when incorrect parameters are returned from a function to the calling routine.

The UNDEFINED FUNCTION Event

If the WHENEVER UNDEFINED FUNCTION statement is in effect, 4GL takes the specified action when it encounters a function call which function has not been defined.

Typically the compiler verifies the functions called within an application to make sure that they have been defined, thus an undefined function will produce a compile-time error in the majority of cases. However, there are two situations in which an undefined function can pass through compilation process and remain undefined at runtime:

- When an application is linked in Lycia Command Line environment and the qlink command has been used without the --verify flag
- When an application is linked against a dynamic library. The function must be defined at the time of linking, however, if the dynamic library is changed later and the function is changed or removed, it will not be reflected in the application, unless you recompile it again

The QUIT Event

The WHENEVER QUIT command is an alternative to the DEFER QUIT statement which provides note flexibility than the DEFER statement. The DEFER QUIT statement effectively translates to:

```
WHENEVER GLOBAL QUIT CALL EventQuitFlag
```

The WHENEVER statement allows you to modify the 4GL reaction to the Quit key being pressed

The INTERRUPT Event

The WHENEVER INTERRUPT command is an alternative to the DEFER INTERRUPT statement which provides note flexibility than the DEFER statement. The DEFER INTERRUPT statement effectively translates to:

```
WHENEVER GLOBAL INTERRUPT CALL EventIntFlag
```

The WHENEVER statement allows you to modify the 4GL reaction to the Interrupt key being pressed

The TERMINATE Event

If the WHENEVER TERMINATE statement is in effect, 4GL takes the specified action when the process is being terminated. This option takes effect only if the process is running on an application server.



The HANGUP Event

If the WHENEVER HANGUP statement is in effect, 4GL takes the specified action when the connection to the client or terminal has closed. This option takes effect only if the process is running on an application server, i.e. only in GUI mode.

FATAL ERROR Event

If the WHENEVER FATAL ERROR statement is in effect, 4GL takes the specified action when a fatal application error has occurred (i.e. segmentation fault).

	Note: When the FATAL ERROR event is returned, the program will always exit. This does, however, provide some simple capability to report/clean up for such conditions.
---	---

WHENEVER Actions

The WHENEVER actions include CONTINUE, STOP, CALL AND GOTO. The CALL action can invoke built-in or programmer defined functions which will be executed when 4GL encounters the specified event. The CALL handler now can invoke some Querix standard processing functions which handle the events. The following functions can be called within the WHENEVER statement:

The WHENEVER is an easy way to instruct a program what should be done, if an exceptional behaviour is spotted, instead of inserting specific 4GL code with the instructions after each statement that may result in an exceptional behaviour.

The WHENEVER ERROR STOP and WHENEVER ERROR CONTINUE statements check the value of the status built-in variable to define whether an error has occurred.

The default action which 4GL undertakes on encountering any of the exceptional conditions is CONTINUE.

The GOTO Option

The GOTO option can transfer the program control to the specified LABEL statement. The GO TO and GOTO statements are interchangeable. The program control transfer is possible only within the same program block and if the *label-name* specified after the GOTO statement is the same as the *label-name* defined by a LABEL statement. In the example below the program control is transferred to the LABEL statement with the name *cannot_delete*, if a NOT FOUND condition occurs:

```
DELETE * FROM contact
      WHERE contact.cont_ord = p_company_id
WHENEVER NOT FOUND GOTO cannot_delete
...
LABEL cannot_delete:
MESSAGE "No such record"
```

In the cases when you have the WHENEVER statement with the GOTO keyword in one program block (i.e. a FUNCTION program block) and it is followed by other FUNCTION blocks within the same program module, the WHENEVER option is also applied to the subsequent program blocks. If the condition specified in the WHENEVER statement is met during the execution of a subsequent program block (i.e. NOT FOUND condition), 4GL tries to transfer the program control to the label, which is not defined in the current program



block. As specified above the GOTO statement can transfer the program control only within the same program block, it cannot transfer control to the LABEL statement which is located in another program block.

You need to redefine the WHENEVER condition. If an SQL statement is processed before it is redefined, an error will occur, if the WHENEVER condition is met. To redefine the condition use another WHENEVER statement (i.e. WHENEVER NOT FOUND CONTINUE) statement, or define the corresponding LABEL within each subsequent program block.

The CALL Option

The CALL option specifies an event handler, which should be invoked when the WHENEVER condition is met. The event handler can be a programmer defined function, built-in function or a special event handler supported by Querix 4GL. No variables can be passed to the function called in such a way, thus the list of arguments either empty or not should be omitted.

```
WHENEVER QUIT CALL err_manag
```

If you use the BEGIN WORK statement in a function called in such a way, always use the WHENEVER ERROR CONTINUE and WHENEVER WARNING CONTINUE before the ROLLBACK WORK statement. Otherwise the program will loop, if the ROLLBACK statement encounters an error or a warning.

You cannot execute a stored procedure by means of the WHENEVER...CALL statement by placing the name of the stored procedure after the CALL keyword. To execute a stored procedure specify a function that contains the EXECUTE PROCEDURE statement after the CALL keyword.

Special Event Handlers

Querix supports additional special event handlers used with the CALL option, they are:

Function	Effect
EventAbort	The process is aborted with an error dialog.
EventTerminate	The program exits immediately with no error.
EventIgnore	The event is ignored and the process is continued. This option will retry all input statements in process rather than exiting them like the WHENEVER... CONTINUE statement.
EventInterruptFlag	Sets the interrupt flag and continues.
EventQuitFlag	Set the quit flag and continues.
EventTerminateDialog	Displays a dialog box requesting permission to terminate. If the user selects yes, the program terminates.

```
WHENEVER INTERRUPT CALL EventInterruptFlag
```

Default Event Handlers

The system installs default global-level event handlers at start-up which take effect when the corresponding event handlers are not specified explicitly. These are as follows:

Event	Default handler
ARGUMENT ERROR	EventAbort
RETURN ERROR	EventAbort
UNDEFINED FUNCTION	EventAbort
INTERRUPT	EventTerminate



QUIT	EventTerminate
TERMINATE	EventTerminate
HANGUP	EventTerminate
FATAL ERROR	EventTerminate
ERROR	EventAbort
WARNING	EventAbort

User Event Handlers

In addition to the standard processing functions the user may write event processing routines. Such routines always return a value. If they return nonzero then the routine is considered not to have processed the event and it is passed to the next level handler. Therefore you can write:

```
FUNCTION work()
    WHENEVER TERMINATE CALL mycleanup
    INSERT INTO mywork VALUES (my_session_id,'an entry')
    CALL mycleanup()
END FUNCTION

FUNCTION mycleanup()
    DELETE FROM mywork WHERE id = my_session_id
    RETURN 1
END FUNCTION
```

This would cause the mycleanup function to be called if the program receives a terminate signal, the program would then try the module-level handler for the terminate signal and then (if that did not handle it) the global one.

The RAISE Option

The RAISE option signals that, in case of an exception, the problem should be handled not by the local function, but by the calling one.

It means that if an exception happens during a function execution after the WHENEVER [ANY] ERROR RAISE statement was encountered, the function execution terminates and the program control returns to the calling routine, i.e., to the statement following the CALL statement. If the calling routine does not contain an error handler, the error is passed further to the parent function.

The RAISE option can be effectively used with the TRY...CATCH statement. You can place the CALL statement to the TRY block and the RAISE option to the invoked function. If any error occurs during the function execution, the program control will pass to the CATCH block:

```
MAIN
TRY
    CALL do_exception(100, 0)
CATCH
    CALL foo()
END TRY
END MAIN
```



```
FUNCTION do_exception(a, b)
    DEFINE a, b, c INTEGER
    WHENEVER ANY ERROR RAISE
    RETURN a / b
END FUNCTION

FUNCTION foo()
    DISPLAY "Exception caught, status: ", STATUS
END FUNCTION
```



Note: The RAISE action cannot be performed in the REPORT program block.

The CONTINUE Option

If the CONTINUE keyword is specified after a WHENEVER condition, 4GL takes no actions when the condition is met. It is the default option for all conditions. Use it to restore the default behaviour.

The STOP Option

The STOP keyword stops the execution of the program when the WHENEVER condition is met. The example below makes 4GL terminate the execution when the database server issues a warning:

```
WHENEVER WARNING STOP
```

The Scope of Reference of the WHENEVER Statement

The WHENEVER statement takes effect from the moment it is encountered by 4GL and to the end of the program module or to the next WHENEVER statement that specifies the same exceptional condition. The keyword ANY does not identify an exception keyword as a unique one, e.g. the WHENEVER ERROR resets either the previously set WHENEVER ERROR or WHENEVER ANY ERROR statement.

```
DATABASE my_database
MAIN
DEFINE int_var INTEGER
WHENEVER ERROR CONTINUE
PRINT "The first row is being inserted."
INSERT INTO my_table VALUES ("first_value", "second value")
WHENEVER NOT FOUND CONTINUE
WHENEVER ERROR STOP
PRINT "The second row is being inserted."
INSERT INTO my_table VALUES ("third_value", "fourth_value")
CLOSE DATABASE
```



```
PRINT "The program is over."  
END MAIN
```

The second WHENEVER ERROR statement overrides the settings of the first WHENEVER ERROR statement, however, it does not influence the WHENEVER NOT FOUND statement.

Querix 4GL supports optional scoping of the WHENEVER statement. Thus there are two types of event handling: global (for the whole program) and module (for a module or a library). The module scoping is the default scoping.

The WHENEVER GLOBAL... statement causes the specified error handler to take effect in the module where it is declared and in all the other modules of the program since it is executed by 4GL. The WHENEVER MODULE... statement causes the specified error handler to take effect in the module where it is declared from the moment it is executed by 4GL. It can be used together with the WHENEVER GLOBAL statement for the same event. E.g. if the user presses the Quit key when 4GL is executing the module where the WHENEVER MODULE QUIT STOP statement is present, the application will be terminated, whereas while executing all the other modules with the WHENEVER GLOBAL QUIT CALL EventQuitFlag statement in effect the application will not be terminated upon pressing this key.



WHILE

The WHILE statement creates a loop which is iterated as long as the Boolean expression specified after the WHILE keyword evaluates as true. The syntax of the WHILE statement is as follows:



Element	Description
Expression	A Boolean expression
Logical Block	A block of executable statements

The statements contained in the WHILE loop are iterated while the condition specified in a Boolean expression is TRUE. 4GL evaluates the Boolean expression, if it is TRUE, all the statements within the WHILE loop (up to the END WHILE keywords) are executed, after that 4GL returns to the beginning of the loop and evaluates the expression once more. If it is TRUE, one more iteration is executed, if it is FALSE, the WHILE loop is terminated and the statements that immediately follow the END WHILE keywords are executed. If the value of the Boolean expression is changed to FALSE during the execution of the loop, the loop still will be executed till the end, it will be terminated only when 4GL evaluates the Boolean expression once more before the next iteration.

In the example below 4GL continues inserting new rows while the *yes_no* variable is evaluated as "y". As soon as the user types "n" in the PROMPT field, the WHILE loop is terminated:

```
LET yes_no = "y"  
WHILE yes_no = "y"  
CALL insert_row()  
PROMPT "Do you want to insert one more row (y/n) : "  
FOR yes_no  
END WHILE
```

The EXIT WHILE and CONTINUE WHILE statements can influence the execution of the WHILE loop. If the database you work with supports transactions, it is advisable that the complete WHILE loop be placed within a transaction.

The Logical Block

The logical block which is included into the IF statement has the following structure:



Element	Description
Declaration Clause	The optional DEFINE statement



Statement List	Executable 4GL or SQL statements
-----------------------	----------------------------------

The WHILE statement can include only one logical block. In its turn the logical block consists of the Declaration clause and the Statement list.

The Declaration clause contains one or more DEFINE statements and is used to declare spot variables. These variables are visible only within this logical blocks and cannot be referenced from outside the WHILE statement. For more information about the [DEFINE](#) statement see the corresponding section of this reference. The DEFINE statement of the Logical Block is optional and can be omitted. For more information about the logical blocks see the "[Logical Blocks](#)" section of this manual.

The Statement list can include:

- any executable 4GL statements which may use the corresponding spot variables.
- SQL statements
- the CONTINUE WHILE keywords
- the EXIT WHILE keywords

The **CONTINUE WHILE** Keywords

If 4GL encounters a CONTINUE WHILE statement, it interrupts the execution of the current iteration of the WHILE loop, comes back to the beginning of the WHILE loop and evaluates the Boolean expression once more. If the expression is true, one more iteration is performed, if it is not, the complete WHILE loop is skipped.

The **EXIT WHILE** Keywords

4GL terminates the WHILE loop and continues execution from the statements that follow the END WHILE keywords, when it encounters the EXIT WHILE keywords. All the statements placed between the EXIT WHILE and END WHILE keywords are skipped.

4GL will not be able to exit the WHILE loop, if the value of the Boolean expression is not changed to FALSE during the execution of the loop, unless the EXIT WHILE keywords are specified. To make 4GL leave the WHILE loop, you can also use other logical transfer of the program control, i.e. the GOTO statement.

The **END WHILE** Keywords

The END WHILE keywords specify the end of the WHILE loop. When 4GL encounters these keywords, unlike in other 4GL statements, it will not execute the statements that follow the END WHILE keywords. 4GL will return to the beginning of the WHILE loop and evaluate the Boolean expression once more. If the expression is TRUE, it will iterate the WHILE loop again. If the expression is FALSE, it will exit the WHILE loop and execute the statements that immediately follow the END WHILE keywords.



WRITE

The WRITE statement is used to record value from the program variables into a file.



Element	Description
Descriptor	The integer variable that stores the file descriptor.
Variable List	The list of program variables separated by commas whose values are to be recorded into the file.

The descriptor should be first linked to a file by means of the OPEN FILE statement. While the file descriptor was opened and not closed, each new WRITE statement appends the values to the end of the file, even if the file was opened without the APPEND option and only with the WRITE option. If the CLOSE FILE was called for the file descriptor followed by another OPEN FILE option, the write statement can behave in two different ways:

- If the second OPEN FILE contains the WRITE option and no APPEND option, the file offset will be set at the beginning of the file and the written values will overwrite the data in the file. It does not rewrite the whole file, it rewrites only that part of the file starting from its beginning that corresponds to the size of the newly written values. If the file is longer than the values written, the remains of the file will be unchanged.
- If the second OPEN FILE contains the APPEND option, the file offset will be set at the end of the file and each new WRITE statement will add new values at the end of the file rather than rewrite existing values.

Like with the READ statement the variable list can contain simple variables, records and arrays. E.g.:

```
WRITE TO file_dscr USING contacts.*  
WRITE TO file_dscr USING var1, var2, var3
```

Variable list

The values of each variable in the variable list will be recorded to the file as an element separated from the rest by the delimiters. The WRITE statement inserts delimiters between values and records them to the file in the order in which they appear in the USING clause. E.g. when writing to a text file, each variable in the USING clause will be a new line.

If the values in the variables contain delimiters, e.g. LET a = "string1 \n string2", then they will be recorded with this delimiter and the variable 'a' in this case will be recorded as two lines into a text file. A backslash symbol



The following example creates a text file and opens it for writing, then records 2 strings into it and closes the file. Next it opens the same file, rewrites the first line with the new value and closes the file again. The third opening is in the append mode, so the WRITE statement writes a third line into the file - the file will contain 3 lines at the end of the operation.

```
MAIN
DEFINE my_f INT
DEFINE s1, s2, s3 STRING

LET my_f = 101
LET s1 = "string1"
LET s2 = "string2"

OPEN FILE my_f FROM "text5.txt" OPTIONS (WRITE, CREATE, FORMAT="text")
    WRITE TO my_f USING s1
    WRITE TO my_f USING s2
CLOSE FILE my_f
LET s2 = "string3"
OPEN FILE my_f FROM "text5.txt" OPTIONS(WRITE, FORMAT="text")
    WRITE TO my_f USING s2
CLOSE FILE my_f
LET s3 = "string4"
OPEN FILE my_f FROM "text5.txt" OPTIONS(APPEND, FORMAT="text")
    WRITE TO my_f USING s3
CLOSE FILE my_f
END MAIN

#at this point the file contents will be:
#string3
#string2
#string4
```



CHAPTER 4:

REPORTS

THIS CHAPTER WILL COVER THE STATEMENTS USED FOR CREATING 4GL REPORTS. IT WILL ALSO TOUCH UPON THE METHODS OF CREATING AND FORMATTING AND OUTPUTTING REPORTS.



4GL Reports Overview

4GL Output Methods

There are several features in Querix 4GL which perform the output of values from 4GL program variables or from a relational database:

- Producing a report from a 4GL program to different output destinations (screen, file, another program, printer) by means of the REPORT program block
- Copying the data from a database into a file by means of the UNLOAD statement
- Producing output to the screen by means of the DISPLAY statement
- Displaying values to a screen form by means of the DISPLAY and DISPLAY ARRAY statements
- Producing output to the reserved lines occupied by the ERROR, MESSAGE PROMPT and MENU statements and to the COMMENT line by means of the COMMENT attribute of a form field
- Producing output of the variables of the large data types (BYTE and TEXT) to an external editor by means of the PROGRAM attribute of a form field

Report Features

Querix 4GL supports a report writer that is used to produce a report from a 4GL program, it has the following features:

- The option that allows you to display the report to the screen for editing
- The features that control the output page layout, they can specify the format of the page headers and footers, the first page of the report, columnar presentation, and special formatting of the output
- The facilities that can create a report from the values returned by the cursor, or from the input records, which assemble values from different places (e.g. from several SELECT statements)
- The facilities that can manipulate the data retrieved from a database for a report either before it is formatted or after.
- The aggregate functions that can perform calculations and produce frequencies, percentages, sums, averages, maxima, and minima
- Built-in functions and operators that enable additional formatting of the report output
- The WORDWRAP operator which formats long character strings
- The option allowing to update a database or to execute a number of SQL or 4GL statements while the report is being processed



Producing 4GL Reports

The report writer can arrange the data in accordance with the instructions in the REPORT program block or in the report driver and to perform the output. The output can be sent:

- to the screen
- to a file
- to a printer
- to a pipe

To be able to write a report a program must have two components:

- The REPORT program block that formats the report output
- The report driver that invokes and processes the report

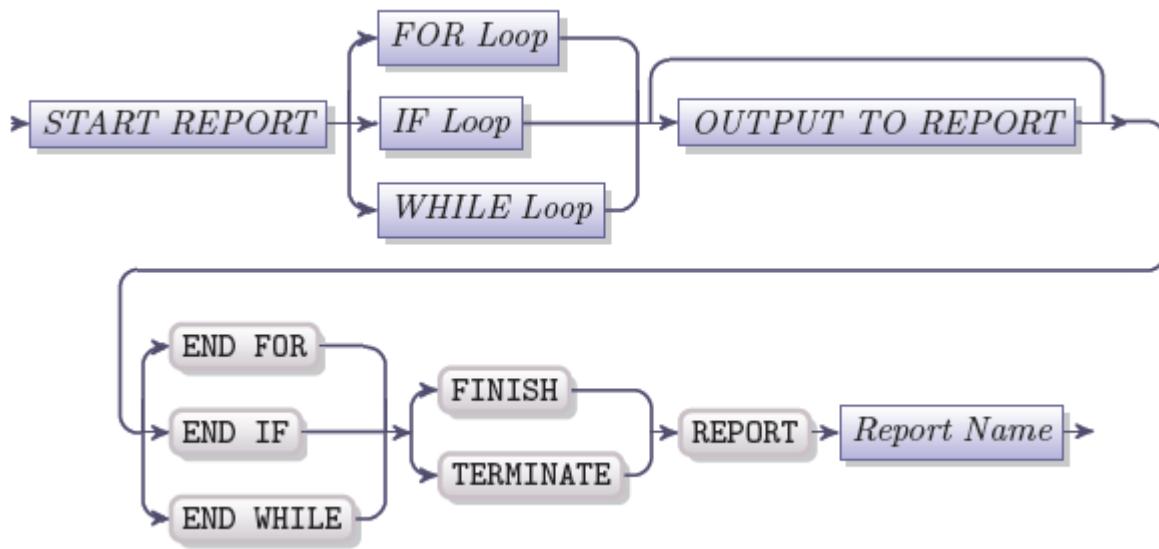
The report driver contains the statements that retrieve rows from a database, store their values in program variables and send them to the report. The information is sent to the report in portions called the input records, after the last such record has been received and formatted by the REPORT program block, the aggregate values based on these records are calculated, if required. After that, the report is sent to the output device.

The fact that the processed of data retrieval and data formatting are separated makes it easier to apply the same report to different sets of data.

The Report Driver

The report driver invokes the report, retrieves the data and sends the retrieved data to the REPORT program block in the form of input records. The data are formatted in the REPORT program block and then sent to the output device.

A report driver cannot be included into the REPORT program block. It can be included into the MAIN program block or a FUNCTION program block. A report driver consists of the following parts: the START REPORT statement, the OUTPUT TO report clause and the FINISH REPORT statement. It can optionally contain the TERMINATE REPORT statement to handle errors that occur during the report processing.



Element	Description
START REPORT	The START REPORT statement with all the optional clauses if necessary
FOR Loop	The FOR statement with all its clauses except for the END FOR keywords
IF Loop	The IF statement with all its clauses except for the END IF keywords
WHILE Loop	The WHILE statement with all its clauses except for the END WHILE keywords
OUTPUT TO REPORT	The OUTPUT TO REPORT statement
Report Name	The name of the report which has been used in the START REPORT statement of this driver

The report driver is usually embedded in a loop (WHILE, FOR, or FOREACH). When 4GL encounters a report driver, it takes the following actions:

- Initializes the report which identifier follows the START REPORT statement
- Begins the loop in which the report driver is embedded, retrieves the rows and stores them in program variables
- Uses the OUTPUT TO REPORT statement to pass the stored values to the REPORT program block
- Terminates the loop after all the values have been retrieved by means of the END FOR, END FOREACH, or END WHILE keywords
- Completes the report by means of the FINISH REPORT keywords: the ON LAST ROW clause is executed, if it is present and the two-pass report processing is activated

4GL can leave the report driver without completing it by means of the TERMINATE REPORT statement. The report is typically terminated by the TERMINATE REPORT statement when 4GL encounters an error. When 4GL encounters the TERMINATE REPORT statement, it does not execute the ON LAST ROW clause.

The REPORT Program Block

The REPORT program block contains the instructions according to which the report output is formatted. It is a separate program block like the MAIN or FUNCTION blocks, this it cannot be include into MAIN, FUNCTION



or another REPORT block. The REPORT program block can have local variables. It cannot be invoked by means of the CALL statement.

The REPORT program block receives the data from the report driver. The data are supplied to it in portions called the input records. Each record is the result of one iteration of a FOR, FOREACH, or WHILE loop within the report driver, it can include program records and other data types including the programmer defined classes. Each input record is formatted independently and sent to the specified report destination. A REPORT program block can contain most of the 4GL functions and statements and can also include special statements and operators available only in a report definition.

The REPORT program block has the following structure:



Element	Description
Report Name	The name you want to assign to the report
Argument List	Formal arguments passed to the report
REPORT Sections	Sections used to format the report output and declare local variables

A typical REPORT program block contains the following components:

- A report declaration section. It contains of the REPORT keyword, the report name, and formal arguments
- Other REPORT sections:
 - A DEFINE section. It declares local variables and formal arguments.
 - An OUTPUT clause which is optional. It specifies the page layout of the output
 - An ORDER BY clause which is optional. It specifies the sorting instructions for the output
 - The FORMAT block. It formats the headers and footers of the report pages and format the data from the input records
- The END REPORT keywords specify the end of the REPORT program block

A 4GL report can process also the data that are not retrieved from the database.

The Report Declaration

The report declaration resembles the function declaration and consists of the report name and the formal arguments enclosed in parentheses.

```
REPORT my_rep (arg1, arg2)
```

A report name has the global scope of reference, it must be unique among other report names, function names and names of the global variables. The name of the report must not conflict with the names of its formal arguments.

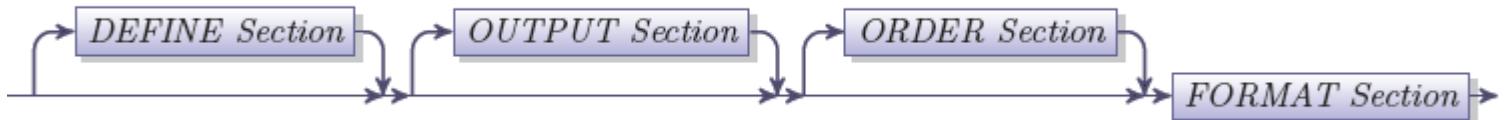
The formal arguments of a report cannot be of the ARRAY data type. They cannot be of the RECORD data type, if the record contains an array as a member. All the formal arguments specified in the argument list must be defined in the DEFINE section of the REPORT program block. If a report has no arguments, an empty argument list must still be present in the report declaration section.



If the argument list is not specified, the output from the report can include text from the control blocks, but the report can include only the data stored in the global and module variables.

The Report Sections

The definition of a report consists of four sections, three of which are optional:



Element	Description
DEFINE Section	Here the local variables are declared
OUTPUT Section	Here the dimensions and other attributes of a report page are set
ORDER Section	Here the rules for sorting the input records passed to the report are set
FORMAT Section	Here the format in which the report information is sent to the report destination is specified

The REPORT program block must include the FORMAT section, it can also contain the DEFINE, OUTPUT, and ORDER BY statements. The first three sections are optional, if they are included, they must be placed in the order in which they are listed above.

Section	Purpose
DEFINE section	The DEFINE is an optional section that declares the variables local to the REPORT program block and the formal arguments passed to it. The DEFINE section is not required in the reports that do not have any formal arguments or local variables.
OUTPUT section	The OUTPUT is an optional section used to specify the margins, the number of lines and the maximum number of characters of each report page. It also specifies the destination of the report output
ORDER BY section	The ORDER BY is an optional section used to sort the values before sending them to an output device. It is required if the report driver does not sort the data which are passed to the REPORT program block
FORMAT section	It is the only required section, which specifies the format of the report including the headers and footers of the report pages as well as the aggregate functions. It can also contain the set of statements that specify the actions which should be performed before or after the specific groups of input records are preceded.

The sections begin with the corresponding keywords (DEFINE, OUTPUT, ORDER BY, and FORMAT). The END keyword is not required to mark the end of a section. One section is ended at the point where the keyword of another section is placed. The sections should be placed in the above listed order.

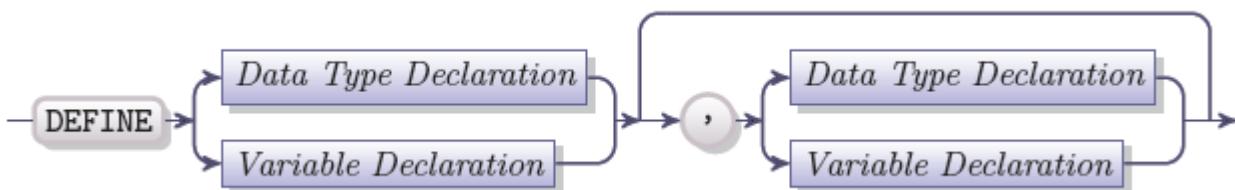
The end of the REPORT program block should be marked with the END REPORT keywords.



DEFINE Section

The DEFINE section declares the data types of the formal arguments of the report as well as the report local variables. The variables declared in the DEFINE section cannot be referenced from outside the REPORT program block. The general rules applied to the DEFINE section are the same as for the DEFINE statement within the MAIN and FUNCTION program blocks. It is not required if the report has no local variables and no formal arguments.

The DEFINE section of the report program block has the same syntax as the DEFINE statement:



Element	Description
Data Type Declaration	A block where a user-defined data type is declared
Variable Declaration	A block where the data type of a variable is declared

For the detailed information about the declaration of variables and data types see the section on the [DEFINE](#) statement of this manual.

The DEFINE report section is used to declare variables local to the REPORT program block and to declare the data types passed to the report by the report driver. In Querix 4GL there are no restrictions regarding the data types of the arguments passed to the report or the data types of the local variables.

The formal arguments are required in the REPORT program block, if one of the following conditions is present:

- The FORMAT EVERY ROW option is specified – all values for each record should be passed to the report as formal arguments
- The ORDER BY section is included – all values in each input record used in the ORDER BY section should be passed to the report as formal arguments
- The AFTER GROUP OF clause is specified – all values in each input record used in this clause should be passed to the report as formal arguments
- If an aggregate appears anywhere in the REPORT program block except in the ON LAST ROW clause, and if this aggregate depends on all records, each of the records should be passed to the report in an argument list

Aggregate that depend on all records include the GROUP PERCENT(*) and any aggregate without the GROUP keyword anywhere in the REPORT program block except in the ON LAST ROW clause.

If an aggregate function contains an argument that is not declared as a report formal argument, an error might occur. However, you can use global and module variables in the aggregate functions and they do not need to be declared in the report DEFINE block. Such arguments should not be changed during the execution of the report.



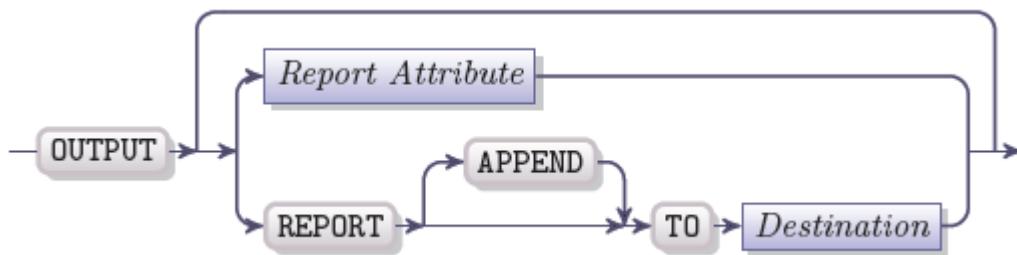
If a variable or a formal argument is declared with the help of the LIKE keyword, you must previously specify the default database by placing the DATABASE statement at the beginning of the program module before the first program block.

A local report variable which name is the same as the name of a global or a module variable will be used instead of such variable within the scope of reference of the REPORT program block. The global and module variables can be referenced within the REPORT program block, however they may cause problems in aggregate functions of the BEFORE GROUP OF and AFTER GROUP OF clauses. The global and module variables can also cause problems in any part of a two-pass report block, if their values change during the report execution.



OUTPUT Section

The OUTPUT section specifies the parameters of the report pages: dimensions of the pages, margins, the page-eject sequence. This clause can be omitted. In that case, the default values for page formatting will be used. The settings specified in this section can be overridden by the settings of the START REPORT section of the report driver.



Element	Description
Report Attribute	An attribute which specifies the dimensions of the report page
Destination	The destination of the report output

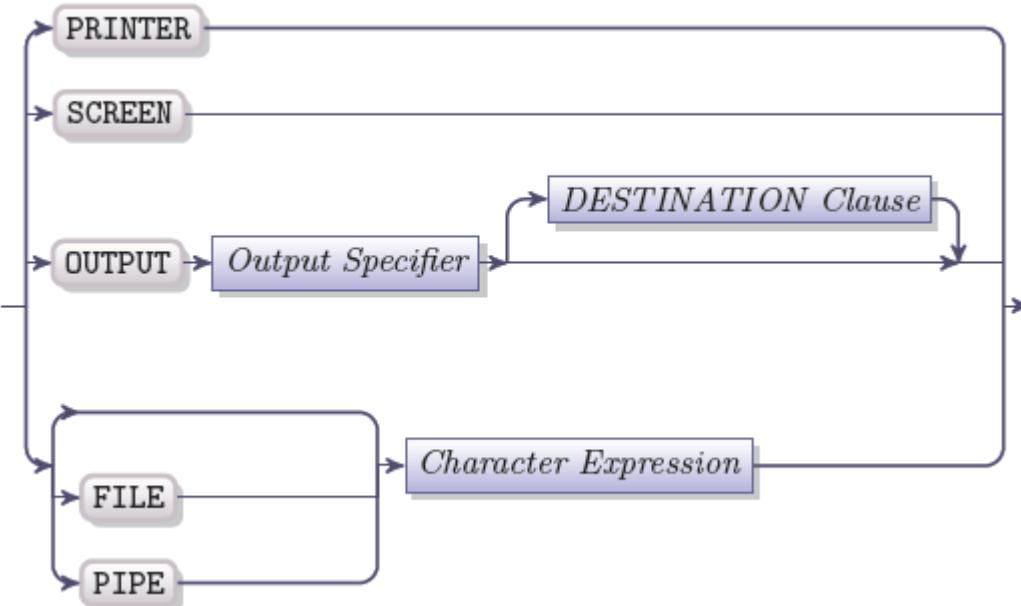
The OUTPUT section does not only specify the page layout but also specifies the destination of the report output. The output can be sent to a printer, file, or pipe. The TO clause of the report driver can override the settings defined by the OUTPUT section. The page layout settings of the OUTPUT clause can be overridden by the settings of the WITH clause of the report driver.

The size of the margins and of the page length can be specified only as literal integer, it cannot be specified with the help of the variables. Thus the START REPORT statement can be used to override these settings and to assign the size dynamically at runtime.

The OUTPUT section begins with the OUTPUT keyword, it must include the REPORT TO keywords which specify the destination of the output. If it is omitted, the output is performed to the screen. If the REPORT TO keywords are followed by the PRINTER keyword, you can specify a 1- or 2-character page-eject sequence in the TOP OF PAGE clause. This sequence causes a printer to begin a new page, rather than to pad the end of each page with blank lines.

Report Destination

The TO clause is optional, it specifies the destination for the output. If this clause is present in the START REPORT statement, it overrides the settings of the REPORT TO clause of the corresponding REPORT program block.



Element	Description
Character Expression	A 4GL character expression that returns a valid destination
Output Specifier	A character string or a variable of character data type
DESTINATION Clause	A clause that specifies a file name or a program name

The TO clause of the START REPORT statement overrides the settings specified in the REPORT TO clause of the OUTPUT section. If the REPORT TO clause is omitted, the output destination for the report is the destination specified in the START REPORT STATEMENT. If there is no TO clause in the START REPORT and no REPORT TO clause in the OUTPUT section of the REPORT program block, the output is made to the screen. Specifying the SCREEN keyword in the REPORT TO clause has the same effect.

If an empty set of data is sent to the report by the OUTPUT TO REPORT clause, or if there is not OUTPUT TO REPORT clause, no output is produced even if you specify the START REPORT CLAUSE, and the TO clause as well as the REPORT TO clause has no effect.

The TO clause can send the output to the following destinations:

- To the screen (if the SCREEN keyword is specified)
- To a printer (if the PRINTER keyword is specified)
- To a file (if the FILE keyword is specified)
- To the command line, another program or shell script (if the PIPE keyword is specified)

The detailed information about the report destination can be found in the section which deals with the START REPORT statement, "[Output Configurations](#)" section.

The APPEND Keyword

The APPEND keyword can be added after the REPORT keyword, if you want to append your report to a previously generated report. This feature is available only if both reports – the new and the old – are output to a file. Appending works only when the output is sent to a file, because reports sent to a printer are always



appended and it is the natural course of things and reports sent to the screen are not overwritten or appended while they are opened in different instances of the Report Viewer.

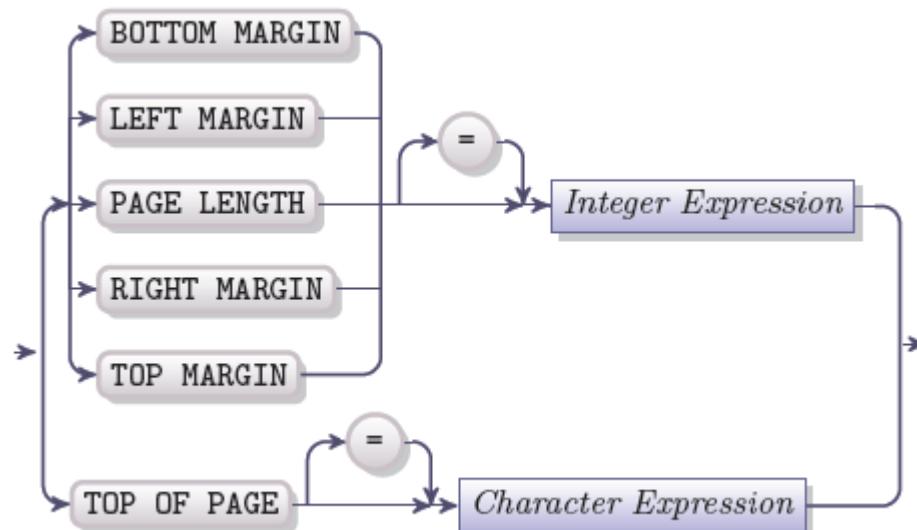
If you omit the APPEND keyword and send a report to a file which contains a previously produced report, the contents of the file will be overwritten by the new report and the data will be lost. If the APPEND keyword is present, the new report will be added to the existing one and appended to the end of the specified report file.

```
START REPORT rep2 APPEND TO FILE "C:\\f_rep.txt"
```

Using the APPEND keyword you can append reports unlimited number of times and no information will be lost. The APPEND keyword can also appear within the START REPORT statement.

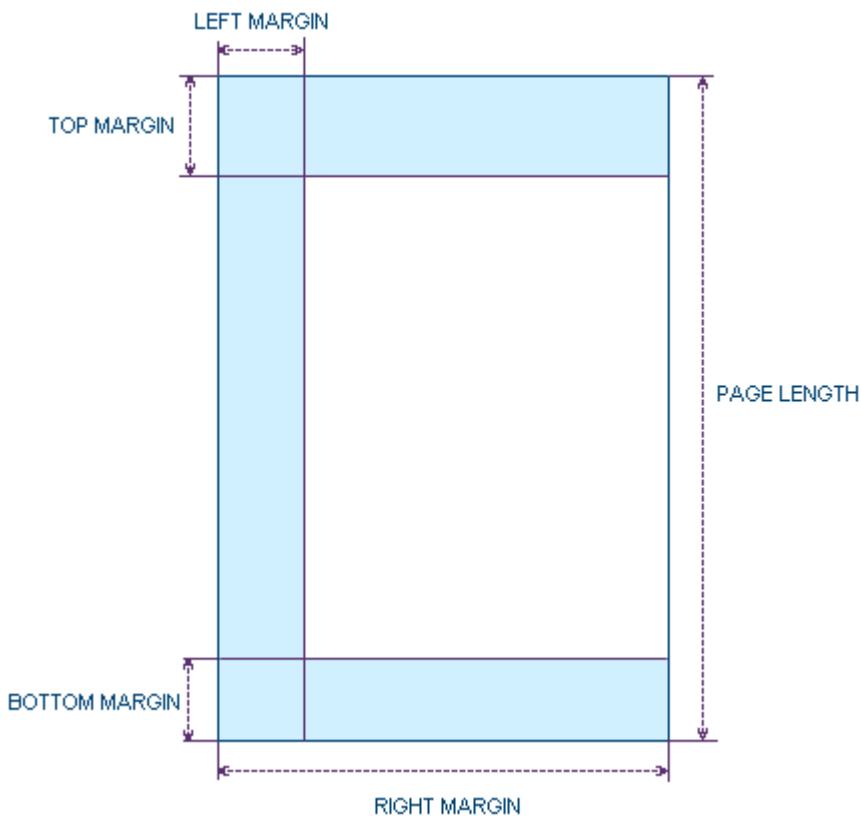
Report Attribute

The report attributes specify the dimensions of the report page. They correspond to the attributes used in the WITH clause of the START REPORT statement and are overridden by this clause, if it is present. The OUTPUT section of the report program block can contain one or more report attributes, they should not be separated by any delimiters. The report attributes have the following syntax:



Element	Description
Integer Expression	An expression returning a positive integer or a literal integer no larger than 2,766.
Character Expression	An expression that returns a quoted character string that specifies the page-eject character sequence

Below is the diagram of a report page, with the positions of all the possible delimiters which can be set in the OUTPUT section:



- The RIGHT MARGIN keywords specify the page width together with the left margin. It can be omitted, if you specify FORMAT EVERY ROW option, then the default value will be used, which is 132 characters
- The LEFT MARGIN keywords specify the number of blank spaces with which each line of the report output should start. By default it is 5 characters
- The PAGE LENGTH keywords specify the page length together with the top and bottom margins as well as the page headers and trailers specified in the FORMAT section. By default it is 66 lines
- The TOP MARGIN keywords specify the number of empty lines that appear at the beginning of each report page above the first line of the actual output text. By default it is 3 lines long
- The BOTTOM MARGIN keywords specify the number of empty lines that follow the last line of the actual output text at the bottom of each report page. By default it is 3 lines long

The values assigned to these parameters cannot be larger than 32,766 and smaller than 1.

The TOP MARGIN Clause

This clause sets the size of the top margin for every page of the report. The default value of the top margin is 3 lines. The output from the PAGE HEADER or FIRST PAGE HEADER defined by the FORMAT section begins at the line following the last line of the top margin, by default it begins in the fourth line.

The following example sets zero value for the TOP MARGIN clause, thus the page header will immediately begin at the top of each page with no empty lines preceding it:

```
OUTPUT
REPORT TO SCREEN
TOP MARGIN 0
```



The BOTTOM MARGIN Clause

This clause sets the size of the bottom margin for every page of the report. The size specified in the BOTTOM MARGIN clause defines the number of empty lines which will appear below the output from the PAGE TRAILER defined by the FORMAT section. If the BOTTOM MARGIN clause is omitted, the default value is three lines. Thus the three lines must be left empty at the bottom of every report page.

The example below specifies zero value in the BOTTOM MARGIN clause, thus no empty lines will be left at the bottom of the report pages:

```
OUTPUT
REPORT TO PRINTER
TOP MARGIN 0
BOTTOM MARGIN 0
```

The PAGE LENGTH Clause

This clause specifies the number of lines contained by each report page. The top and bottom margins are included into the page length value. The default value is of the page length is 66 lines. On a standard 24-lines video screen, 22 lines is the maximum number of lines which can be used together with the PAUSE statement to avoid undesirable scrolling.

```
OUTPUT
REPORT TO SCREEN
PAGE LENGTH 22
```

The page length should be longer than the sum of the values assigned to the top and bottom margins and the sizes of the header and trailer of the page. The above mentioned parts of the report page cannot display the data of the report, thus a page which page length is equal to this sum is not available for displaying data.

The LEFT MARGIN Clause

This clause sets the size of the left margin for each line of the report. The output of each line begins at the *value+1* position. The size is measured in characters. Measurements specified by the COLUMN function are always relative to the margin set by the LEFT MARGIN clause. The default value for the left margin is 5 characters, thus the output of each report line begins in the sixth character position by default.

The example below makes 4GL start each line of the report in the third character position:

```
OUTPUT
REPORT TO "my_file"
LEFT MARGIN 2
```

The RIGHT MARGIN Clause

This clause sets the size of the right margin for each line of the default report or of a PRINT WORDWRAP statement. The default report is the report which has the EVERY ROW clause in the FORMAT section. Unlike other clauses that specify margins, the RIGHT MARGIN clause does not specify the number of characters which should remain blank at the right side of the page. It specifies the total width of a page in characters (the width of every line in a page).



The size of the right margin does not depend on the size of the left margin. However, the right margin includes the left margin and the right margin starts count from the leftmost character of a line.

The default size of the right margin is 132 characters, but the default size is applied only if the FORMAT section of the REPORT program block contains the EVERY ROW specification, or if the PRINT WORDWRAP statement is executed. If one of these conditions is in effect, to apply the default size of the right margin you should omit the RIGHT MARGIN clause.

A default report which contains the EVERY ROW specification places the variable names at the top of the page and then places their values below in columns. If the width of the default page is not sufficient and the variables cannot be placed in one line, a two-column output is performed. 4GL lists the names of the variables and the data of each output record on each line of the output.

The example below specifies that the page width (RIGHT MARGIN) should be 100 characters. Though the EVERY ROW specification is present in the FORMAT section, the default size of the right margin is not applied, because the RIGHT MARGIN clause is included.

```
REPORT my_report(id, name)
DEFINE id LIKE lang.id
      name LIKE lang.name
OUTPUT
TOP MARGIN 5
BOTTOM MARGIN 1
RIGHT MARGIN 100
FORMAT
EVERY ROW
END REPORT
```

Temporary Line Width

If the FORMAT section includes the PRINT statement with the WORDWRAP RIGHT MARGIN clause, this clause specifies a temporary margin. The temporary margin cannot be larger than the right margin set by the RIGHT MARGIN clause in the OUTPUT section.

The temporary settings of the right margin override the settings of the OUTPUT section during the execution of the PINT statement. After the execution of the PRINT statement is complete, the right margin settings of the OUTPUT section are restored.

The TOP OF PAGE Clause

This clause specifies the page-eject character sequence for a printer. The page-eject character specified in this clause is used by the 4GL to set up new pages. In the example below the CONTROL-L is specified as the page-eject character:

```
OUTPUT
TOP OF PAGE "^L"
REPORT TO "rep_file"
```



On many printers “^L” (the ASCII form-feed character) is the page-eject character. If you are not sure which string is used as the page-eject character of your printer, refer to the printer documentation. If the TOP OF PAGE clause contains a character string which begins with any other symbol except the caret (^), the first character of this string is considered to be the page-eject character. If the first character of the character string is the caret (^), the second symbol is interpreted as the control character.

If the TOP OF PAGE clause is included into the OUTPUT section, the page breaks in the output are initiated by using the specified page-eject character and not by padding with blank lines. If this clause is not included, the LINEFEED characters are used to extend the page length to the specified size.

A New Report Page

Before a new page can be processed, 4GL extends the previous page to the specified page size by means of padding blank lines or by using the page-eject character. A new page is started when:

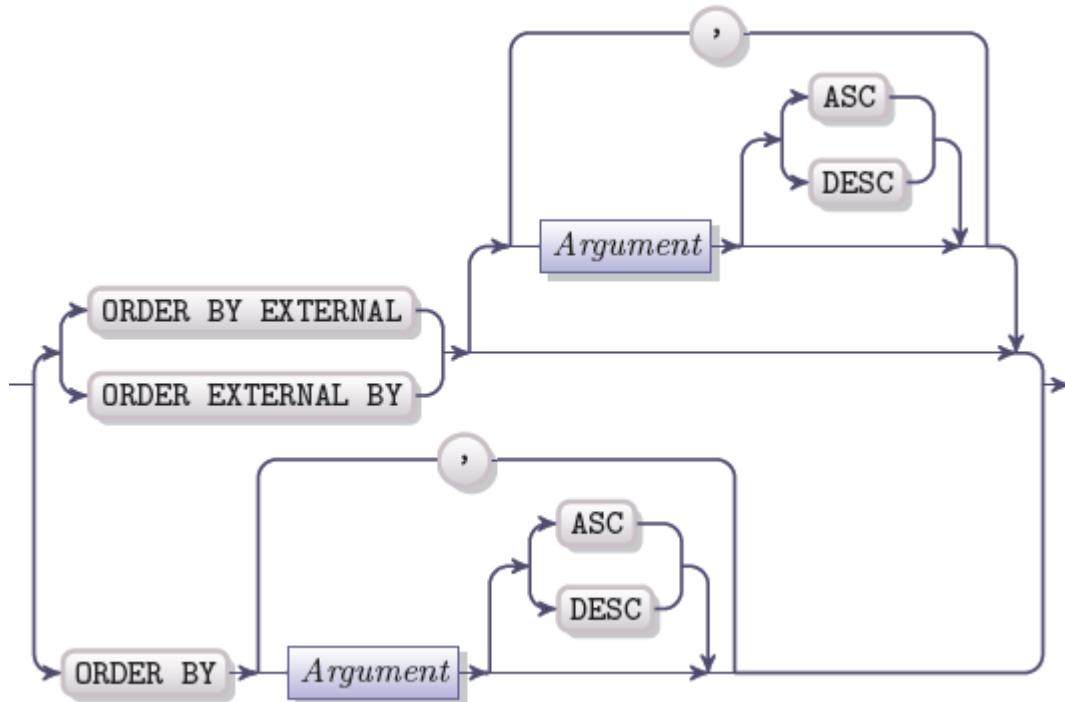
- The PRINT statement attempts to print on a page that is already full
- The NEED statement specifies the number of lines that exceeds the number of the remaining lines on the current page
- The SKIP TO TOP OF PAGE statement is executed
- The SKIP...LINES statement is executed and the number of lines specified in this statement is more than the number of the available lines on the current page

If the TOP OF PAGE clause is not included, the remaining lines of the current page are filled with the LINEFEED characters, when a new page is initiated.



ORDER BY Section

The ORDER BY section sorts the input records passed by the report driver to the REPORT program block. It determines the sequence of execution of the GROUP OF control clauses in the FORMAT section.



Element	Description
Argument	The name of an argument from the report declaration (as described in " The Report Declaration ")

The ORDER BY clause specifies the sorting criteria for the input records. The ORDER BY section should be included, if values received from the report driver are used in determining how BEFORE GROUP OF or AFTER GROUP OF control blocks will process the data.

If this section is omitted, the input records are processed in the order in which they are received by the REPORT program block. The GROUP OF clauses are processed in the order of their appearance in the FORMAT section. The GROUP OF clause can be executed at random intervals after any input record, if the report driver does not sort the input records, because unsorted values usually change from record to record.

If the GROUP OF clause contains only one variable, and the values have already been sorted in sequence on that variable by the ORDER BY clause of the SELECT statement, the ORDER BY section is not required.



The Sorting List

The ORDER BY clause contains the list of variables which specify the order in which the input records must be sorted. Only those variables can be used in the sorting list of the ORDER BY clause, which have been declared as the formal arguments of the report.

```
REPORT my_report(x,y)
DEFINE x RECORD LIKE table.*,
y INT
ORDER BY y
```

If the ORDER BY section contains more than one variable, they are used for sorting in the left-to-right sequence of variables as the order of decreasing precedence. The default is the ascending arrangement (which can be explicitly specified by means of the ASC keyword), the descending arrangement can be applied by specifying the DESC keyword. The values are sorted by the first variable in the sorting list. The values which have the same value for the first variable are sorted by the second variable in the list and so on.

The sorting based on the variables of the character data types is based on the code-set character order, unless the DBNLS environment variable is set to 1 and a non-default collation sequence is defined.

The DESC keyword indicates that the values will be ordered descending from the highest to the lowest. However the precedence of the variables in the sorting list is the same as in the case of the ascending sorting – the values are sorted by the variables in the left-to-right order.

```
ORDER BY var2, var5 DESC
```

The values can also be sorted by means of specifying the ORDER BY clause in the SELECT statement. If the ORDER BY clause is specified in the SELECT statement and the REPORT program block contains the ORDER BY section, the ORDER BY section of the report definition takes precedence.

If all the values for the report are retrieved with the help of one cursor, the report is processed faster, provided that the values are sorted by the ORDER BY clause of the SELECT statement.

Even if the values are sorted by the report driver, you may specify the ORDER EXTERNAL BY option to specify the exact order of the GROUP OF control clauses processing.

The Sequence of the GROUP OF clauses Execution

The ORDER BY section can be used to determine the order in which the BEFORE GROUP OF and AFTER GROUP OF clauses are processed. These clauses are processed in the physical order of appearance, if the ORDER BY clause is omitted.

If the ORDER BY clause is included and there are more than one BEFORE GROUP OF or AFTER GROUP OF clause, the order of the execution of these clauses is determined by the sorting list of the ORDER BY clause. In this case the physical order of these clauses in the FORMAT section does not influence the order of their execution.

The statements contained in a BEFORE GROUP OF or AFTER GROUP OF clause are executed:



- Every time the value of the current group variable is changed
- Every time the value of a higher-priority variable is changed

The update rate of a group variable depends on whether the records have been sorted. If the records are sorted, 4GL executes the AFTER GROUP OF clause after the last record of the specified group is processed. The BEFORE GROUP OF clause is executed before the first record with the same value for the group variable is processed.

If the records are not sorted, the BEFORE GROUP OF and the AFTER GROUP OF clauses may be executed before and after each record is processed respectively, because the value of the group variable may change after each record is processed. If the ORDER BY clause is omitted, the BEFORE GROUP OF and the AFTER GROUP OF clauses are executed in the order of appearance.

The EXTERNAL Keyword

The EXTERNAL keyword can be used in the ORDER BY clause, if the input records have been sorted by the ORDER BY clause of the SELECT statement. The sorting list that follows the ORDER EXTERNAL BY keywords defines the order of the GROUP OF clauses execution. The ORDER BY EXTERNAL can be used as a synonym.

If the EXTERNAL keyword is absent from the ORDER BY section, a report will be a two-pass report and 4GL will have to process the input records twice. During the first pass the input records are received, sorted and stored in a temporary file. During the second pass the aggregate values are calculated and the output is sent to the output device. If the EXTERNAL keyword is not included, the report requires access to a database where it can store the temporary file. If no database is open when the report is being processed, a runtime error occurs.

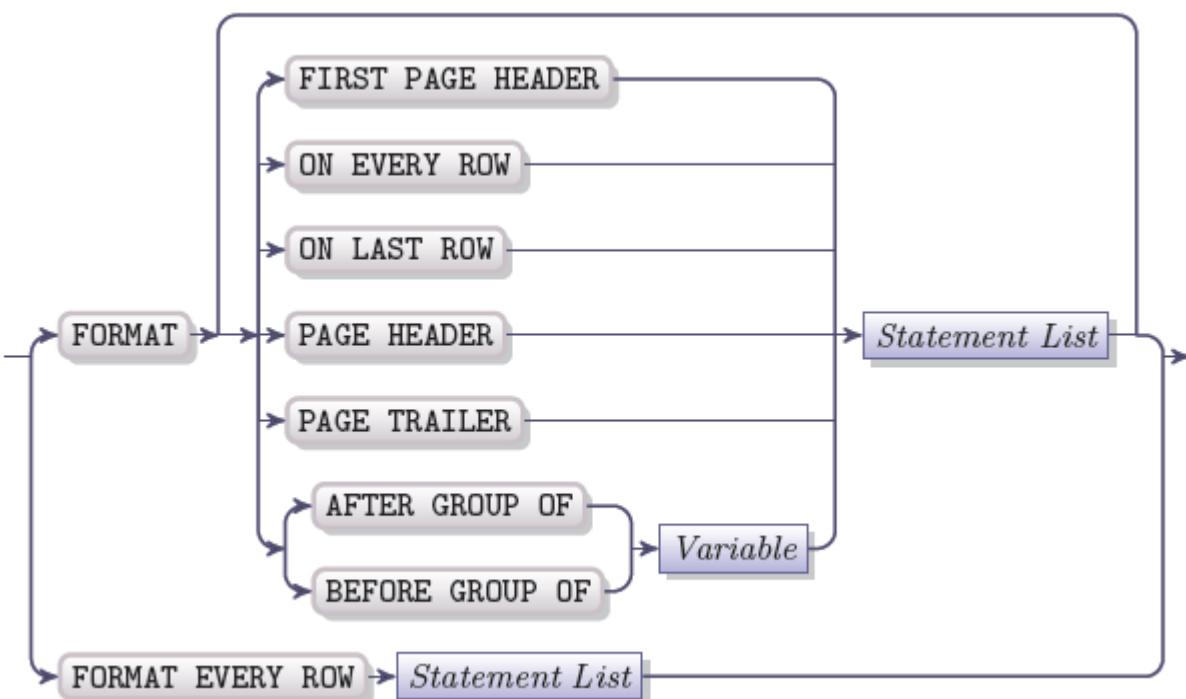
If the EXTERNAL keyword is present, the report is a single-pass report. 4GL does not require temporary files to process such report. If you use the ORDER EXTERNAL BY section in your REPORT program block, it may result in the performance improvement. If the EXTERNAL keyword is included, the report does not require access to a database.

If the input records are passed to the REPORT program block already sorted because they are all retrieved by means of one cursor, use the ORDER BY clause in the SELECT statement and then use the ORDER EXTERNAL BY in the REPORT program block.



FORMAT Section

A REPORT program block must contain the FORMAT section, it is the only obligatory section of the block. The FORMAT section defines how the output if formatted. It operates the values passed to the report as formal arguments as well as with the global and module variables. The FORMAT section is always the last section of the REPORT program block, it begins with the FORMAT keyword and ends together with the block with the END REPORT statement. It has the following syntax:



Element	Description
Statement List	A list of 4GL statements including report execution statements described in " Executable Statements in the REPORT Program Block "
Variable	The name of a formal argument to the report definition. You must pass at least the value of variable through the arguments.

There are two types of the FORMAT section which can be used in the REPORT program block:

- A FORMAT section that contains the EVERY ROW keywords. It should have only the FORMAT keywords, the ON EVERY ROW clause and the END REPORT keywords. This FORMAT section specifies the default report.
- A complex FORMAT section that contains control blocks (i.e. ON EVERY ROW, BEFORE GROUP OF, AFTER GROUP OF, etc.) with executable statements (these cannot be SQL statements)

There are seven types of the FORMAT control clauses and execution statements that can be used in the FORMAT section.



If the EVERY ROW keywords are not included into the FORMAT section, you can combine one or more control clauses in any order within the FORMAT section. The BEFORE GROUP OF and AFTER GROUP OF control blocks can appear several times in one FORMAT section, all the other FORMAT control clauses can be used only once per FORMAT section.

The EVERY ROW Clause

The EVERY ROW keywords are used to create a default report, it includes every input record that is proceeded by the REPORT program block. If the FORMAT section includes the EVERY ROW clause, no other FORMAT control clauses can be used.

The default report contains only the values passed to it by the report driver. No additional values can be included into the report, no executable statements can be used in a default FORMAT section. The values are passed to the default report through its formal arguments.

If you want the report output to be displayed in a format that differs from the default format, use the ON EVERY ROW control clause.

Below is an example of a default report with the EVERY ROW keywords specified in the FORMAT section:

```
REPORT my_report (mytable)
DEFINE my_table LIKE mytable.*
FORMAT
    EVERY ROW
END REPORT
```

The names of all the variables passed to the report by the report driver are used as the column headings in the report output. If the headings cannot fit into one horizontal line, the names of the variables are printed down the left side of the page. The values are printed down the right side of the page:

```
my_table.column1 1000
my_table.column2 2000
my_table.column3 3000
my_table.column4 4000

my_table.column1 9998
my_table.column2 9997
my_table.column3 9996
my_table.column4 9995
```

If the columns headings can fit in one line, the column headings are printed on the first horizontal line and the values are printed below them in columns. A default report has the following structure:

First_name	Last_name	Age	Gender
John	Smith	32	M
Jane	Brown	28	F
William	Green	54	M

When a variable contains a NULL value, the variable name is printed as the column name, but nothing is printed for its value.

The RIGHT MARGIN keywords can be used in the OUTPUT section to define the overall width of the report page of the default report.



FORMAT Control Clauses

The control clauses of the FORMAT section are used to define the structure of a report. A FORMAT control clause contains one or more executable statements, these statements are executed when the certain parts of the report are being processed.

If no input records are passed to the REPORT program block, none of the FORMAT control clauses are executed. This means that if the report driver does not contain the OUTPUT TO REPORT clause or if this clause does not pass at least one input record to the REPORT program block, the control clauses of the FORMAT section are not executed.

There are seven FORMAT control clauses, they all are optional, but a FORMAT section that does not include the EVERY ROW keywords must contain at least one control clause.

A FORMAT section can contain the following control clauses:

- FIRST PAGE HEADER – is processed before the first report page
- PAGE HEADER – is processed before every new report page
- BEFORE GROUP OF – is executed before the specified group of the sorted records
- ON EVERY ROW – is executed as each row is passed to the REPORT program block
- AFTER GROUP OF - is executed after the specified group of the sorted records
- PAGE TRAILER – is processed after each report page is complete
- ON LAST ROW – is processed after the last input record has been passed to the REPORT program block

The BEFORE GROUP OF and AFTER GROUP OF control clauses can reference the same variable. If the value of such variable changes and if the ON EVERY ROW clause is included, the report processes all BEFORE GROUP OF clauses before the ON EVERY ROW clause as well as the ON EVERY ROW clause is processed before every AFTER GROUP OF clause.

The order of execution of the BEFORE GROUP OF and AFTER GROUP OF clauses depends on the order in which the input records passed to the report are sorted. The records are sorted by the sorting list of the ORDER BY section. Thus if an ORDER BY section specifies the following sorting list: var1, var2, var3, these clauses will be executed in the following order:

```
BEFORE GROUP OF var1 -- 1
    BEFORE GROUP OF var2 --2
        BEFORE GROUP OF var3 --3
            ON EVERY ROW -- 4
                AFTER GROUP OF var3 --3
                    AFTER GROUP OF var2 --2
                        AFTER GROUP OF var1 --1
```

In the example above, a clause can be executed a multiple number of times relative to any other block that is marked with a lower number in the comment.

The physical order of the clauses in this example does not influence the sequence of execution, if the ORDER BY section contains the corresponding sorting list. If the ORDER BY section is absent, the order of execution matches the physical order of the clauses.

If some variables acquire new values in the PAGE HEADER clause, the new values are not available until after the first SKIP, PRINT, or NEED statement is executed in an ON EVERY ROW control clause. This means that the values printed by means of the PAGE HEADER clause have the same values as they have in the ON EVERY ROW clause.

Statements that Cannot Be Used in the FORMAT Section

Some executable statements cannot be used within the FORMAT control clauses, these are:

CONSTRUCT INPUT ARRAY



DEFER	MAIN
DEFINE	MENU
DISPLAY ARRAY	PROMPT
FUNCTION	REPORT
INPUT	RETURN

If any of these statements is included into a control block of the FORMAT section, a compile-time error will occur. If you need to use these statements, include them into a function and use the CALL statement in the FORMAT control clause to invoke this function.

The AFTER GROUP OF Clause

The statements contained in the AFTER GROUP OF control clause are executed after a group of sorted records has been processed. The sorting of the records is performed according to the ORDER BY section of the REPORT program block, or by the ORDER BY clause of the SELECT clause.

DIAGRAM

The AFTER GROUP OF clause must be followed by a group variable which should match a formal argument of the report. A group of records is all the input records that have the same value for the formal argument, specified in the AFTER GROUP OF clause. You cannot use a variable that has not been passed to the report as a formal argument in this clause. Each formal argument can have only one AFTER GROUP OF clause referring to it. When the statements of this clause are executed, local variables have the values from the last record of the current group.

The Order of Processing

An AFTER GROUP OF clause is executed in the following cases:

- When the value of the group variable is changed
- When the value of a variable which has higher priority in the sorting list is changed
- After the last input record has been processed, but before the ON LAST ROW or PAGE TRAILER clauses have been executed (the AFTER GROUP OF clause is executed in ascending order in this case)

The sorting list of the ORDER BY section specifies the order in which the input records are sorted. How often the value of the group variables changes depends on whether the records have been sorted by the SELECT statement.

If the input records have been sorted by the ORDER BY clause of the SELECT statement before they are passed to the REPORT program block, the AFTER GROUP OF control clause is executed after the last input record of the group has been processed.

If the input records are passed to the REPORT program block unsorted, the AFTER GROUP OF clause can be executed after any input record has been processed, due to the fact that the value of the group variable can be changed with each new record. The AFTER GROUP OF clauses are executed in the physical order in which they are listed in the FORMAT section, if no ORDER BY section is included.

However, the AFTER GROUP OF and BEFORE GROUP OF clauses are designed for working with sorted data. The records can be sorted either by the report driver, if the SELECT statement contains the ORDER BY clause, or by the ORDER BY section of the REPORT program block.

To sort the input records in the report driver before sending them to the REPORT program block, follow these steps:

- Specify the ORDER BY clause in the SELECT statement
- Use the name of a column after the ORDER BY keywords (the same way the name of the group variable is used after the AFTER GROUP OF clause)



- Specify the ORDER EXTERNAL BY keywords in the ORDER BY section of the REPORT program block, if the FORMAT section contains a BEFORE GROUP OF or AFTER GROUP OF clause.
- Specify the precedence of the variables in the sorting list that follows the ORDER EXTERNAL BY keywords.

To sort data with the help of the ORDER BY section, the name of a formal report argument must be specified in the AFTER GROUP OF clause and in the sorting list of the ORDER BY section. If the input records are sorted both in the report driver and in the REPORT program block, the sorting of the REPORT program block takes precedence.

If the sorting list contains more than one variable, the records are sorted by the first variable which has the highest priority. The records which have the same value for the first variable are sorted by the second variable. The records which have the same values for all the variables except the last one are sorted by the last variable which has the lowest priority.

The GROUP Keyword

The GROUP keyword can be used in the AFTER GROUP OF clause to qualify aggregate report functions (such as avg(), min(), max(), sum()). The AFTER GROUP OF clause is the only valid FORMAT control clause where the GROUP keyword can be used.

```
AFTER GROUP OF var1
    PRINT 20 SPACE, "_____"
    SKIP 1 LINE
    GROUP SUM(total_val) USING "$$$,$$$,$$$,&&"
```

The aggregate functions are discussed in details later in this chapter

The FIRST PAGE HEADER Clause

The statements included into the FIRST PAGE HEADER clause are executed before 4GL processes the first input record. It can be used to specify the information which should be printed at the top of the first page of the report, or other instructions that should be executed before the report is started. E.g. it can be used to initialize the variables used in the REPORT program block before the output is performed.

In the example below the FIRST PAGE HEADER is used both to display the title of the report and to perform other activities (initialize a variable and call a function):

```
FIRST PAGE HEADER
    LET a=1
    CALL get_it_rec (it_ord.rec_id) RETURNING r_rec.*
    PRINT COLUMN 30, "The List of Items"
    SKIP 1 LINE
    PRINT "_____"
    SKIP 2 LINES
```

The FIRST PAGE HEADER clause is not executed as well as the other FORMAT control clauses, if the report driver does not pass at least one input record to the REPORT program block.

Displaying Titles and Headings

The FIRST PAGE HEADER can be used either for executing some statements before the report is sent to the output device, or for displaying some information at the top of the first page of the report. The information can include the title of the report, the column headings, etc. This control clause overrides the settings of the PAGE HEADER clause for the first page of the report. If the FIRST PAGE HEADER is absent, the PAGE



HEADER specifies the output for the first page as well, whereas the FIRST PAGE HEADER does not define the headers of all the subsequent pages that follow the first one in case the PAGE HEADER clause is omitted. The TOP MARGIN clause specifies the number of empty lines at the top of the first page which cannot be occupied by the output from the FIRST PAGE HEADER clause or the PAGE HEADER clause. The output from these clauses begins immediately after the last empty line.

Statements Restrictions

Some statements cannot be used in the FIRST PAGE HEADER control clause, or some restrictions are applied to their usage:

- The NEED statement cannot be used
- The PRINT *file-name* statement cannot be used
- The SKIP integer LINES statement cannot be used in this clause, if it is included into a loop
- If the IF statement is used and the THEN and ELSE keywords are followed by the PRINT statements, the number of lines displayed by such PRINT statements must be equal
- The PRINT statement in the CASE, FOR, or WHILE statements must be terminated by means of the semicolon symbol (;). The semicolon symbol is needed to suppress the LINEFEED characters which may appear due to the loops, it keeps the number of lines constant for every report page

These restrictions apply for the statements used not only in the FIRST PAGE HEADER clause, but also in the PAGE HEADER and PAGE TRAILER clauses.

The ON EVERY ROW Clause

The statements included into the ON EVERY ROW control clause are executed each time an input record is passed from the report driver to the REPORT program block. This clause is often used to print the core report data. The example below prints the values of the variables on each row, each input record passed to the report changes the values of these variables:

```
ON EVERY ROW
PRINT item_id, COLUMN 10, item_name, COLUMN 30, item_desc
```

The processing of the PAGE HEADER or FIRST PAGE HEADER control blocks is suspended until 4GL encounters the first ON EVERY ROW clause which contains a PRINT, NEED, or SKIP statement.

Group Control Clauses

If the value a group variable in a BEFORE GROUP OF clause changes, all the relative BEFORE GROUP OF clauses are executed before the ON EVERY ROW clause is processed. The execution of the AFTER GROUP OF clause is also triggered by the change of the group variable value and all the relative AFTER GROUP OF clauses are executed before the ON EVERY ROW clause.

The ON LAST ROW Clause

The statements included into the ON LAST ROW control clause are executed after the last input record passed to the REPORT program block has been processed and when 4GL encounters the FINISH REPORT statement in the report driver.

The statements in the BEFORE GROUP OF and AFTER GROUP OF clauses are executed before the ON LAST ROW clause is processed, if these clauses are present.

While the ON LAST ROW clause is being executed, the variables retain the values of the last input report. You can use the ON LAST ROW control clause to specify aggregate functions which will display the total values, the minimum or maximum value, etc.



```
ON LAST ROW
SKIP 1 LINE
PRINT "_____"
PRINT "Total sum of orders: ", COLUMN 35, count(*) USING "$$$,$$$,$$$,&"
```

If the processing of the report is terminated by means of the TERMINATE REPORT statement rather than FINISH REPORT statement, the ON LAST ROW clause is not executed.

The PAGE HEADER Clause

The statements included into the PAGE HEADER control clause are executed before every new report page is produced. It can be used to print some information at the top of each page.

```
PAGE HEADER
PRINT "Left Column", COLUMN 30, "Right Column" COLUMN 35, PAGENO
```

The PAGE HEADER clause is executed each time a new report page is initiated. The TOP MARGIN specification of the OUTPUT section defines the number of empty lines at the top of each page which will precede the output from the PAGE HEADER clause.

The PAGENO operator can be used in the PRINT statement of this clause to display the number of the current page. The settings of the PAGE HEADER control clause are overridden by the FIRST PAGE HEADER clause for the first page of the report.

The execution of the PAGE HEADER clause is suspended until 4GL encounters the first PRINT, SKIP, or NEED statement within either ON EVERY ROW, BEFORE GROUP OF, or AFTER GROUP OF clauses. It is done because new group values may appear in the PAGE HEADER clause, thus such suspended execution guarantees that the values printed by the PAGE HEADER and by the ON EVERY ROW clauses are the same. The restrictions in the statements usage described above for the FIRST PAGE HEADER clause apply also for the PAGE HEADER and PAGE TRAILER clauses.

The PAGE TRAILER Clause

The PAGE TRAILER clause can be used to display some information at the bottom of each report page. This clause is executed after all the rows for the current page have been processed but before the PAGE HEADER in the newly initiated page is executed.

A new report page can be initiated in one of the following ways:

- The PRINT statement cannot send the output to the current page because it is already full
- The SKIP *integer* LINES statement specifies more lines than are available on the current report page
- The NEED statement specifies more lines than are available on the current page
- The SKIP TO TOP OF PAGE statement is encountered by 4GL

The PAGENO operator can be included into a PRINT statement contained by the PAGE TRAILER clause to display the number of the current report page at the bottom of the page.

```
PAGE TRAILER
PRINT COLUMN 60, PAGENO USING "Page <<<"
```

The value of the BOTTOM MARGIN defines how many empty lines will follow the output from the PAGE TRAILER clause.



The PAGE TRAILER cannot produce different number of lines from page to page, their number must be consistent throughout the report and it must be unambiguously expressed. The restrictions applied to the CASE, FOR, IF, NEED, SKIP, PRINT, and WHILE statements used in the FIRST PAGE HEADER clause is true for the PAGE TRAILER clause as well.



Executable Statements in the REPORT Program Block

The executable statements can be used in the control clauses of the REPORT program block. The control clauses are located in the FORMAT section and they define when 4GL should take the actions specified by the executable statements which these clauses contain.

The list of statements in a control clause begins after the keywords what specify the name of this control clause and ends when 4GL encounters either the keywords specifying the beginning of another control clause, the END REPORT, or the EXIT REPORT keywords.

The following statements can be used in the REPORT program block:

- Most of the 4GL statements (except those which are listed above in the subsection "Statements that Cannot Be Used in the FORMAT Section")
- The majority of the SQL statements
- Special 4GL statements

The 4GL statements which are widely used in the FORMAT section are statements that create loops: CASE, IF, FOR, WHILE. The LET statement is also often used. Their syntax in the REPORT program block is the same as in any other part of a 4GL program.

The special 4GL statements can be used only in the FORMAT section of the REPORT program block, they are EXIT REPORT, NEED, PAUSE, PRINT, SKIP. They cannot be used in other program blocks (i.e. FORMAT or MAIN).

If the executable statements located in the REPORT program block reference local variables, these variables have to be declared in the DEFINE section of the report.

Special Statements of the FORMAT Section

The statements listed below are valid only in the FORMAT section of the REPORT program block:

- EXIT REPORT – when 4GL encounters this statement, the report execution is terminated
- NEED – when 4GL encounters this statement, a new page is invoked, unless the number of available lines is greater than the number of lines specified in the NEED statement, or equal to it
- PAUSE – when 4GL encounters this statement, it suspends the report output until the RETURN key is pressed
- PRINT – when 4GL encounters this statement, it sends the values that follow this statement to the output of the report
- SKIP - when 4GL encounters this statement, it inserts the specified number of empty lines into the report or initiates a new page.



EXIT REPORT

When 4GL encounters the EXIT REPORT statement, it terminates the report and returns the program control to the first statement that follows the most recently executed OUTPUT TO REPORT statement.

When 4GL encounters the EXIT REPORT statement, it performs the following actions:

- Terminates the report processing
- Deletes temporary files and tables if they have been created during the processing of the report

The EXIT REPORT statement used in the REPORT program block produces the same effect as the TERMINATE REPORT statement used in the report driver. They are not interchangeable. If the TERMINATE REPORT statement is included into the REPORT program block, it can only reference the report driver of another report.

Use the EXIT REPORT statement to terminate the report in case an error or other problem prevents the report from producing a portion of information.

If 4GL encounters the EXIT REPORT statement, it does not execute the ON LAST ROW clause if it is present. It also does not format values from the aggregate functions. The EXIT REPORT produces an empty report if the input records are sorted by means of the ORDER BY section of the REPORT program block, unless this section contains the EXTERNAL keyword.

The RETURN statement is not valid in the REPORT program block and thus it cannot be used as a substitute for the EXIT REPORT statement.



NEED

The NEED statement is used to specify a number of lines required for output of a portion of information on a single page, when the splitting of such information is undesirable. It causes 4GL to initiate a new page, if there are fewer available lines at the bottom of the report page than the number of lines specified in the NEED statement. Thus you can prevent the dividing of the output parts that should logically stick together.



Element	Description
Integer Expression	The integer expression which returns a positive integer specifying the number of free lines required

The integer that specifies the number of lines must be followed by the keyword LINES (LINE is a synonym for LINES, they are interchangeable).

The NEED statement can be substituted by the SKIP TO TOP OF PAGE statement in a conditional statement (i.e. IF statement), the condition being that the value of the integer expression is greater than the number of available lines at the bottom of the page.

In the example below the NEED specifies that the following portion of the output requires at least 10 free lines. If there are fewer than 10 lines at the bottom of the page, a new page is initiated.

```
AFTER GROUP OF group_var
      NEED 10 LINES
      PRINT 3 SPACES it_val, COLUMN 10, GROUP sum(r.tot_val)
```

If the number of remaining lines is smaller than the number of lines specified in the NEED statement, the PAGE TRAILER clause of the current page and the PAGE HEADER clause of the new page are processed. The number of remaining lines does not include the bottom margin.



PAUSE

The PAUSE statement is used to suspend the output from the report. It has no effect unless the output is sent to the screen and only if the 4GL application is run under Linux/Unix OS. On Windows, the PAUSE statement produces no effect even if the output is sent to the screen.

The PAUSE statement on Linux/UNIX

If the PAUSE statement is included, the report output remains on the screen until the RETURN key is pressed.



Element	Description
String Expression	A character expression that returns the message displayed by the PAUSE statement

The string expression following the PAUSE statement can be a quoted character string or any 4GL expression that returns such string.

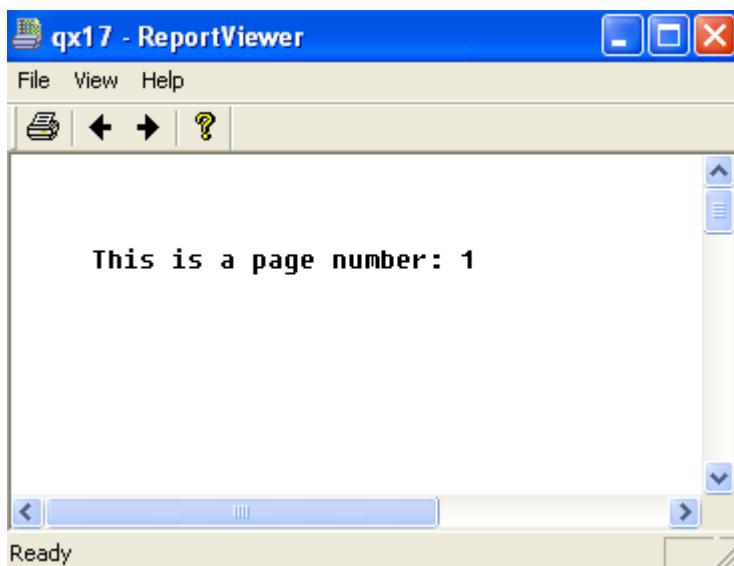
If the TO clause is present in the START REPORT statement and is not followed by the SCREEN keyword, the PAUSE statement has no effect. Likewise, if the OUTPUT section contains the REPORT TO keywords and they are not followed by the SCREEN keyword, the PAUSE statement does not influence the output.

The PAUSE statement is typically included into the PAGE HEADER or PAGE TRAILER clause. In the example below it is included into the PAGE HEADER clause, thus the new page of the report will be displayed only after the user presses the RETURN key.

```
PAGE TRAILER  
PAUSE "Press RETURN to view the next page"
```

Output to the Screen on Windows

Querix has designed the Report Viewer, it is a tool which is a part of the graphical clients. This tool is used for displaying the output sent to the screen. It contains the complete report, the pages can be viewed by means of the left and right arrow buttons on the toolbar of the Report Viewer. Thus you can view any page of the report at any time.



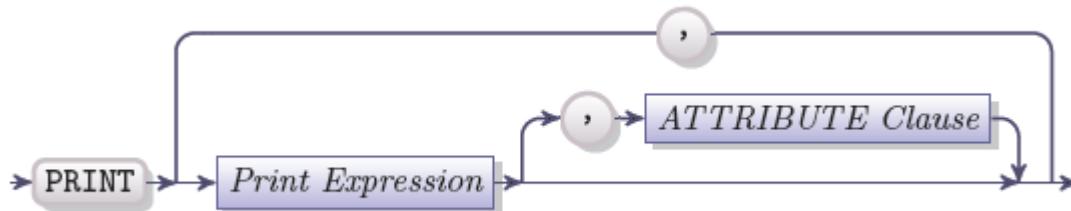
If the report is produced by the application run in the character mode, the output will be sent to the Notepad. All the pages are placed in one text file which is opened automatically and all the pages of the report will be available at the same time.

The PAUSE statement may be omitted, if the application will be run only on Windows OS. However, if it is to be run both on Windows and on Linux/UNIX, the PAUSE statement can be included. It will not trigger an error and will be ignored when the application is run on Windows. However, the message contained in the PAUSE statement will never be displayed on Windows.



PRINT

The PRINT statement is used to send data to the output device from the REPORT program block.



Element	Description
Print Expression	A 4GL expression the value of which is sent to a report
ATTRIBUTE Clause	The display attributes of the value sent to a report

The PRINT keyword indicates the beginning of the list of expressions which are to be sent to the output. If the list contains more than one value, the values must be separated by commas.

The variables of the BYTE data type cannot be displayed by means of the PRINT statement, the "<byte value>" string will be displayed instead. All the variables used in the 4GL expressions must be declared in the DEFINE section of the REPORT program block, provided that they are not declared as module or global variables.

The output produced by the PRINT statement is sent to the output destination specified in the REPORT TO section of the OUTPUT section or in the TO clause of the report driver. In the example below both character expressions and variables are used in the PRINT statements:

```
PAGE HEADER
PRINT COLUMN 15, "MY REPORT", COLUMN 20, PAGENO
ON EVERY ROW
PRINT var1, COLUMN 10, var2 CLIPPED, COLUMN 20, var3 USING "###&", COLUMN
10, var5 CLIPPED, COLUMN 10, "Total: ", COLUMN 2, tot_var
```

ATTRIBUTE Clause

Each print record can have its own ATTRIBUTE clause which specifies the display attribute for the printed value. This clause supports all the standard display attributes of colour and intensity described for the common [ATTRIBUTE clause](#). The ATTRIBUTE clause should be separated from the print statements by commas, otherwise a compile-time error will occur.

The attributes specified in the ATTRIBUTE clause apply only to the print expression preceding this clause. In the example below the "g_id" will be printed bold and "tot_price" will be printed in red, whereas all the other print expressions will be printed normal:



```
PRINT g_id, ATTRIBUTE(BOLD), 2 SPACE, item_name, 2 SPACE, item_quant, 2  
SPACE, it_price, COLUMN 70, tot_price, ATTRIBUTE(RED)
```

Changing the font size

There is one attribute specific to the PRINT statement: FONTSIZE. This attribute sets the font size for the print expressions preceding the ATTRIBUTE clause. It must be followed by the integer, specifying the font size, e.g.:

```
PRINT g_id, ATTRIBUTE(FONTSIZE 14, RED)
```

The Print Expression

The Print expression is an expression which returns a value sent to the report. One PRINT statement can have more than one print expression, they should be separated by commas. It can be one of the expressions listed below:

- 4GL expression – the value returned by the expression is printed.
- COLUMN *integer* – where *integer* returns a positive integer specifying the character position left offset, it cannot be a greater value than the difference (right margin - left margin).
- PAGENO – returns the number of the current report page
- LINENO – returns the number of the currently printing line
- *integer* SPACE / *integer* SPACES – where *integer* returns a positive integer specifying the number of whitespaces between the last symbol of the previous value and the first character of the next one
- Aggregate report functions – functions that return the total value resulting from a number of input records within a report
- *character_expression* WORDWRAP – where *character_expression* is a long character string or a TEXT variable which will be automatically wrapped onto successive lines
- TEXT *variable* – where *variable* is the text variable the value of which will be printed
- FILE "filename" – where *filename* is the name of a text file which must be printed

The FILE Keywords

When 4GL encounters the FILE keywords in the PRINT statement, it reads the contents of the specified file and places them into the report output at the current position of the cursor. The PRINT statement can display only those files which exclusively contain ASCII characters. In the example below the PRINT FILE statement prints the contents of a text file:

```
PRINT "Description of ", item_name  
PRINT FILE "C:/descript_item.txt"
```

The PRINT statement can print the contents of the file without the FILE keyword, if the content of the file is assigned to a variable of the TEXT data type. Such variable together with the variables of other data types can be used in the PRINT statement. In the example below the *desc_itm* is the name of a TEXT variable:



```
PRINT "The functions of the selected item are: ", desc_itm
```

The latest variant can include the WORDWRAP operator, whereas the PRINT FILE statement cannot.

The Character Position

The output from the PRINT statement begins at the current character position. The initial default character position for each new report page is the first character of the first line. This default position can be moved by means of specifying the TOP MARGIN and by the PAGE HEADER and FIRST PAGE HEADER clauses, while the initial character position cannot be located within the margin or page header.

The current character position can also be affected by the following statements:

- SKIP – moves the current character position down the specified number of lines
- NEED – moves the current character position to the new page if the number of available lines on the current page is not sufficient
- PRINT – can move the current cursor position horizontally and sometimes down

The width of the values of different data types has the default value, the USING and CLIPPED operators can be used to override the default settings which are as follows:

Data Type	Default Width Displayed in Characters
BYTE	12 (the actual value of the variable will not be displayed, the <byte value> string will be displayed instead)
CHAR	The declared size
DATE	10
DATETIME	From 3 to 25 depending on the declared data type
DECIMAL	2+p where p is the declared precision
FLOAT	14
INTEGER	11
INTERVAL	From 3 to 25 depending on the data type declaration
MONEY	3+p where p is the declared precision
SMALLFLOAT	14
STRING	The assigned size
TEXT	6
VARCHAR	The declared size

Each PRINT statement performs output on a new line, unless the FILE or WORDWRAP keyword is specified. The example below performs output on three lines, because three PRINT statements are used:

```
ON EVERY ROW
```



```
PRINT fname CLIPPED, 1 SPACE, lname CLIPPED
PRINT address
PRINT "The phone number is: ", p_num
```

The Expression List

The expression list is the list of one or more 4GL expressions following the PRINT keyword that return printable characters. This list can include some built-in functions and operators. Some of them can be used exclusively in the REPORT program block.

Built-in Functions & Operators	Description	Restrictions
ASCII	Converts an operand of the INTEGER data type into the corresponding ASCII character.	
avg()	Returns the average value. (It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block, or in SQL statements. The percent(*) aggregate cannot appear in such SQL statements.
CLIPPED	Discards the trailing blanks from a character operand.	
COLUMN	Specifies the horizontal position of the first character of the operand on the current line.	
count(*)	Returns the total number of records. (It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block, or in SQL statements. The percent(*) aggregate cannot appear in such SQL statements.
CURRENT	Returns the current time and date from the system clock.	
DATE	Converts operands of CHAR, VARCHAR, STRING, or DATETIME data type to the DATE data type	
date()	Returns the current date and day of week	
day()	Returns a positive integer, corresponding to the day indication of the value of its operand (which must be of DATE or DATETIME data type).	
extend()	Converts a DATE or DATETIME value used as operand to the DATETIME value of default or declared precision.	
GROUP	(It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block
length()	Returns the number of bytes stored in its argument of character data type.	
LINENO	Returns the number of the line which is currently printing.	Can be used only in the FORMAT section of a REPORT program block.
max()	Returns the maximum value. (It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block, or in SQL statements. The percent(*) aggregate cannot appear in such SQL statements.



min()	Returns the minimum value. (It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block, or in SQL statements. The percent(*) aggregate cannot appear in such SQL statements.
mdy()	Returns a DATE value from three integers used as operands.	
month()	Returns the number of the month (integer from 1 to 12) from a DATE or DATETIME operand.	
ord()	Returns the integer value of the first byte of the character expression used as an operand.	
PAGENO	Returns the number of the current page of the report.	Can be used only in the FORMAT section of a REPORT program block.
percent(*)	Returns the percentage of the total number of records. (It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block.
SPACES/SPACE	Returns a string of blank characters (ASCII 32), the length of such string is specified by the preceding I positive integer.	
sum()	Returns the total value. (It is described later in this chapter)	Can be used only in the FORMAT section of a REPORT program block, or in SQL statements. The percent(*) aggregate cannot appear in such SQL statements.
TIME	Converts the time portion of a DATETIME operand to a character string.	
TODAY	Returns the current date from the system clock.	
UNITS	Converts an integer expression to an interval.	
USING	Defines the format of the output of the numeric, DATE and MONEY values.	
weekday()	Returns a positive integer which corresponds to the day of week implied by the DATE or DATETIME value used as an operand	
year()	Returns a positive integer which corresponds to the year specified in the DATE or DATETIME value used as an operand	

The USING operator takes precedence over the settings of the DBDATE, DBMONEY, and DBFORMAT environment variables.

Aggregate Report Functions

The aggregate report functions are used to summarize the data from several input records processed by the report. The syntax and effects of these functions resemble the SQL aggregate functions, but they are not identical.



The argument of the sum(), avg(), min(), or max() functions placed between the parentheses is usually of numeric or INTERVAL data type. The values of ARRAY, BYTE, RECORD, and TEXT data types are not valid. These aggregate functions ignore the input records in which these arguments have null values. If the argument has null value in all the input records of the report, the function returns NULL.

The GROUP Operator

The GROUP keyword placed before an aggregate function causes this function to include the data only for the records which have the same value for the group variable specified in the AFTER GROUP OF clause. The GROUP keyword is only valid within the AFTER GROUP OF clause.

The WHERE Clause

The WHERE clause containing a Boolean expression can be placed after an aggregate function if you want to select the records for which the Boolean expression is TRUE among all the input records passed to the REPORT program block. The WHERE clause is optional.

The avg() and sum() Functions

The avg() aggregate function calculates the arithmetic mean value among all the input records processed by the report or of all the records selected by the where clause if it is present. The sum() aggregate function calculates the sum total of all the input records passed to the report or of all the records selected by the where clause if it is present.

The count(*) and percent(*) Functions

The count(*) aggregate function returns the total number of records selected by the WHERE clause. The percent(*) aggregate function calculates the percentage of the total number of the input records in the report. The asterisk symbol (*) must be included.

In the example below the count(*) function is used to calculate the total number of ordered items and the sum() function is used to calculate the total price of the ordered items, both of them use only those records which conform to the WHERE condition:

```
AFTER GROUP OF item_id
PRINT COLUMN 5, "The number of the selected items: ", count(*) WHERE
item_sel MATCHES "yes"
SKIP 1 LINE
PRINT 5 SPACES, "The sum total of the order",
sum(ord_price) WHERE item_sel MATCHES "yes"
```

The min() and max() Functions

The min() and max() aggregate functions return the minimum and maximum values for the numeric, MONEY and INTERVAL values among all the input records passed to the report, or among the records selected by an optional WHERE clause and any GROUP specification.

These functions can also calculate the maximum and minimum values for the values of DATE, DATETIME and character data types. The minimum value in relation to time data types means the earliest and the maximum means the latest. If these functions are used to sort values of character data types of the default locale (U.S. English) the minimum means the first character in the sequence where 1 < A < a and the maximum means the last character in this sequence. If the COLLATION category in the locale defines a non-default collation sequence and the DBNLS variable is set to 1, the default collation sequence has no effect.



The ASCII Operator

The ASCII operator should be followed by the integer expression that returns a positive integer which specifies the code of the character to be printed.

The ASCII operator works with the PRINT statement in the same manner as it works throughout the 4GL code with one exception. The ASCII operator followed by a zero in PRINT statement will produce a NULL character in the report output, whereas if combined with other statements the "ASCII 0" expression will produce a blank space.

The COLUMN Operator

The COLUMN operator should be followed by the integer expression that returns a positive integer. This integer specifies the character position offset. It moves the current character position to the right on the current line of the report page. The positive integer following the COLUMN operator cannot be greater than the total width of the report page excluding the left margin width.

The below example places the current cursor position at the first character of the current line which is the default position:

```
COLUMN 1
```

If the operand of the COLUMN operator is smaller than the current character position, this operator is ignored.

The operand of the COLUMN operator is specified as the number of characters between the left side of the report page (excluding the left margin) and the current character position that is set. Unlike the SPACE operator it does not specify the number of empty spaces between two subsequent expressions if the PRINT expressions list. The COLUMN operator specifies the current character position relative to the left side of the report page, whereas the SPACE operator specifies the current character position in relation to the position of the last character of the previous character expression.

The two PRINT statements below have identical effect:

```
PRINT 2 SPACE, "345", 1 SPACE, "78", 1 SPACE, "10"  
PRINT COLUMN 2, "345", COLUMN 6, "78", COLUMN 9, "10"
```

The LINENO Operator

The LINENO operator does not take an operand, it returns a positive integer which corresponds to the number of the current line of the report page which is being printed. The number of the current page is calculated by counting the number of the lines from the top of the current page including the lines of the top margin. The value returned by this operator can be used for conditional displaying of information:

```
IF LINENO > 9 THEN  
    PRINT "It in the tenth line of the report page"  
END IF
```

The PAGENO Operator

The PAGENO operator does not take an operand, it returns a positive integer which corresponds to the number of the current report page. The next example illustrates the usage of the PAGENO operator in the PAGE HEADING, which prints the number of the current page each time the new page is initiated:



```
PAGE HEADER  
PRINT COLUMN 40, PAGENO USING "Page &&&"
```

The PAGENO operator can be used in the PAGE TRAILER clause or in other clauses to number the report pages.

If the number of input records on each report page is fixed and known and the count(*) aggregate function in the SELECT statement is used to define how many records are returned by the query, the total number of report pages can be calculated. If the total number of pages is known, you can set the page numbering in the format that specifies the number of current page of the total number of pages.

```
SELECT COUNT(*) num FROM customer INTO TEMP temp_table  
SELECT * FROM customer, temp_table  
...  
FORMAT  
FIRST PAGE HEADER  
LET x = temp_table/30  
...  
PAGE TRAILER  
PRINT "Page ", PAGENO USING "<<< " " of ", x USING "<<<"
```

In the example above the total number of pages of the records returned by the query is written down into a temporary table. The total number of pages is defined by dividing the total number of records by the number of record allowed per page (30). The total number of pages has to be rounded up, if there is a reminder. If the total number of pages is 100, the first page will be numbered as

Page 1 of 100

and the last one will be

Page 100 of 100

The SPACE Operator

The SPACE operator must be preceded by an integer expression that returns a positive integer specifying an offset from the current character position. It should be no greater than the difference (*right margin – current character position*). The SPACES is the synonym of SPACE keyword, they are interchangeable.

This operator returns a string of whitespaces, it has the same effect as a quoted string that consists of the same number of blanks. The spaces are inserted at the current character position, when 4GL encounters the SPACE statement. after the PRINT *integer* SPACES statement has been executed, the current character position is moved to the right by the specified number of white spaces.

This operator can appear outside the PRINT statement (i.e. in the DISPLAY statement) where it performs the same function and is used according to the same pattern:



```
FORMAT
ON EVERY ROW
LET variable1 = 5 SPACE, "=ZIP"
PRINT fname, 2 SPACES, lname
```

The SPACES operator together with its argument can be included into parentheses outside the PRINT statement, but the parentheses are optional.

The WORDWRAP Operator

The WORDWRAP operator is used to automatically wrap character strings which are too long onto successive lines of the report. The too long lines are the lines which are longer than the number of characters remaining between the current character position and the right margin of the report page. Such character string is divided into segments and displayed between the current character position and the temporary right margin. If the temporary RIGHT MARGIN is not specified, the RIGHT MARGIN size on the OUTPUT section is used, or it defaults to 132 characters, if the RIGHT MARGIN is not specified explicitly in the OUTPUT section.

Use the WORDWRAP RIGHT MARGIN to set a temporary right margin which will be counted beginning from the left side of the page. The value of the temporary right margin cannot be smaller than the current character position and greater than 132 (or the size of the right margin specified in the OUTPUT section).

The current character position becomes the temporary left margin. The temporary margin values override the margin values from the OUTPUT section or the default margins. The margins of the OUTPUT section or the default margins are restored after the PRINT statement has executed.

The following PRINT statement sets the temporary left margin to 10 and the temporary right margin to 50:

```
PRINT COLUMN 10, my_text WORDWRAP RIGHT MARGIN 50
```

Tabs, Line Breaks, and Page Breaks

The print expression can include the following symbols:

- ASCII printable characters
- TAB symbol (ASCII 9)
- LINEFEED symbol (ASCII 10)
- ENTER symbol (ASCII 13)

Other nonprintable symbols may result in runtime errors. If a data string cannot fit between the margins, it is broken at a word division. The rest of the line is padded with blank spaces at the right.

The TAB characters are expanded from left to have the number of blank spaces which is enough to reach the next tab stop. The tab stops are placed in every eighth column by default, beginning at the left-hand edge of the page. If the next tab stop or a string of blank characters exceeds the right margin the following actions are taken:

1. The blank characters are printed only to the right margin
2. The remaining blanks are discarded
3. The new line is started at the position of the temporary left margin
4. The next word is printed



A new line is started, if a word together with the next whitespace cannot fit the current line. A character string can be broken in the following places (in descending order of precedence):

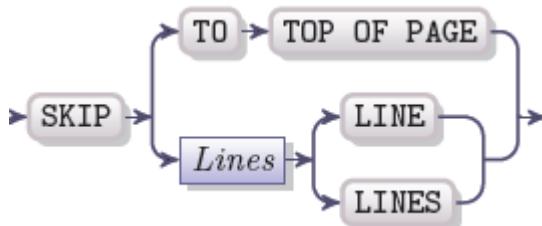
- Break the character string at any LINEFEED, or ENTER, or LINEFEED, ENTER pair
- Break the character string at the last blank character (ASCII 32) or TAB character before the right margin
- Break the character string at the right margin if no character further to the left is a blank space, TAB, ENTER or LINEFEED character.

If the character string is too long for being displayed in the current report page, 4GL prints it to the bottom of the current page, executes PAGE TRAILER and PAGE HEADER clauses and continues printing the string on the next report page.



SKIP

The SKIP statement can be used only in the FORMAT section of the REPORT program block. It inserts the specified number of empty lines into a report, or skips the remainder of the current report page and starts the new one.



Element	Description
Lines	An integer expression that specifies the number of empty lines to be inserted

The SKIP statement is used to advance the current print position to the next page of the report or to move it down by a specified number of lines. The SKIP keyword can be followed by:

- The TO TOP OF PAGE keywords – the current print position is moved to the top of the next page
- An *integer* LINE structure, which specifies the number of lines by which the current printing position should be moved downwards. The LINES and LINE keywords are interchangeable in this context.

When the printing position is moved, the output produced by the PAGE HEADER and PAGE TRAILER clauses remains in its usual position.

```
...
PAGE HEADER
PRINT COLUMN 35, "ORDER INFO"
SKIP 2 LINE
PRINT "ORDER ID", COLUMN 10, "ORDERED BY", COLUMN 30, "ITEMS",
COLUMN 50, "QUANTITY", COLUMN 58, "PRISE", COLUMN 70, "SUM TOTAL"
```

The SKIP statement cannot be used within the CASE, FOR, or WHILE statement. The SKIP TO TOP OF PAGE statement cannot be used in the PAGE HEADER, PAGE TRAILER and FIRST PAGE HEADER clauses.



CHAPTER 5: PRE- PROCESSOR

THIS CHAPTER WILL COVER THE STATEMENTS USED FOR CREATING 4GL REPORTS. IT WILL ALSO TOUCH UPON THE METHODS OF CREATING AND FORMATTING AND OUTPUTTING REPORTS.



Pre-Processor

You can use the pre-processor to transform your source code before compilation. The pre-processor allows you to define different kinds of macros and to include other files. In general, 4GL pre-processor behaves very similar to the C Pre-processor, though it has some peculiarities.

When the pre-processor is invoked, it reads the source file and splits it into lines. Then the lines which are a part of the pre-processor definition are merged together into a single line without removing comments unless they are included into the macro definition. Each such line is then split into a list of lexical tokens.

The pre-processor can do the following:

- Include files.
- Perform conditional compilation.
- Define and expand macros:
 - Simple macros
 - Function macros

The pre-processor directives start sign is the dollar sign (\$) by default, so the pre-processor directives will have the following format:

```
$<directive name>
```

The comments which are produced by the pre-processor have the following format:

```
$LINE number "filename"
```

Here:

- number - the current line number of the processed file.
- filename - the name of the currently processed file.

Invoking the Pre-Processor

The preprocessor is invoked before the file is compiled using the following command:

```
qxpp [-? | -I | -D | -F | -S]
```

This command can use the following flags:

-?	the help flag displaying the information about the usage of the command.
-I path --include-path path	the flag that sets the directory for the included files. Must be followed by the path to the directory.
-D arg --definite arg	the specified definition will be set to the provided argument.
-F filename --infile filename	sets the name of the input file
-S --sharp-beginsign	the start directives sign is the dollar sign (\$) by default. Set this flag to change the sign to the sharp sign (#).



File Inclusion

One of the pre-processor options is file inclusion. To instruct the pre-processor to include a file use the following command:

```
$include "file name"
```

The file name refers to a file that is to be included. The preprocessor searches for this file in the directory of the currently processed file or in the directory for the included files set by the -I flag of the qxpp command. Once this instruction is encountered, the referenced file will be opened and processed and after that the processing of the rest of the current file will be continued.

The example below shows how the included file is added to the pre-processor output file:

Line No.	Processed Source file "a.4gl"	Line No.	Included File "b.4gl"	Line No.	Pre-Processor Output File
0	MAIN	0	DISPLAY	0	\$LINE 1
1		1	"something	1	"a.4gl"
			"		
0	\$includ	0		0	MAIN
2	e	2		2	
	"b.4gl"				
0	END	0		0	\$LINE 1
3	MAIN	3		3	"b.4gl"
0		0		0	DISPLAY
4		4		4	"something
			"		
0		0		0	\$LINE 3
5		5		5	"a.4gl"
0		0		0	END MAIN
6		6		6	



Note: One and the same file can be included more than once, but the recursive inclusions will cause an error.

Defining Macros

There are several types of macros that can be understood by the pre-processor. All of them begin with the following pre-processor directive:

```
$define
```



Defining a Simple Macro

A simple macro definition consists of the macro name and macro body. It has the following syntax:

```
$define name body
```

Here, the name is the name of the macro and the body is a sequence of tokens until the end of the line.

	Note: A simple macro replaces only one line of code, so even if the macro body consists of more than one line, they are merged into a single line as a result. A macro body taking up more than one line requires backslash symbols (\) at the end of each line.
---	---

When pre-processor moves through the code, it looks for the macro name in the source code and replaces the names with the macro body. After the substitution the macro definitions are removed and replaced with blank lines.

Line No.	Processed Source file "a.4gl"	Line No.	Pre-Processor Output File
01	\$define my_macro "Hello world!"	01	\$LINE 1 "a.4gl"
02	MAIN	02	
03	DISPLAY my_macro	03	MAIN
04	END MAIN	04	DISPLAY "Hello world!"
05		05	END MAIN

The pre-processor scans the file line by line, so a macro takes effect from the place where it was included into the file. It has no effect on the lines of code above it, e.g.:

Line No.	Processed Source file "a.4gl"	Line No.	Pre-Processor Output File
01	MAIN	01	\$LINE 1 "a.4gl"
02	DISPLAY my_macro	02	MAIN
03	\$define my_macro "HELLO!"	03	DISPLAY my_macro
04	DISPLAY my_macro	04	
05	END MAIN	05	DISPLAY "HELLO!"
06		06	END MAIN

A macro body is expanded only when the macro is applied in the code:

Line No.	Processed Source file "a.4gl"	Line No.	Pre-Processor Output File
01	\$define macro1 macro2	01	\$LINE 1 "a.4gl"
02	\$define macro2 "Hello!"	02	
03	DISPLAY macro1	03	
04		04	DISPLAY "Hello!"



The recursive expansion of macros is not allowed to prevent infinite recursion. Thus if macro A is expanded to B and B is expanded to A in its turn, A is not expanded to B again, because it appears in its own expansion. Also if the marker is expanded to itself, it can be expanded only once and the recursive expansion does not occur.

Line No.	Processed Source file "a.4gl"	Line No.	Pre-Processor Output File
01	\$define A B	01	\$LINE 1 "a.4gl"
02	\$define B A	02	
03	\$define C C	03	
04	DISPLAY A	04	
05	DISPLAY C	05	DISPLAY A
06		06	DISPLAY C

Defining a Function Macro

Function macros differ from simple macros due to the fact that they accept arguments. Function macros have the following syntax:

```
$define name([argument1 [, argument2...]]) body
```

Here, name is the name of the macro and body is a sequence of tokens until the end of the line. Argument list can consist of a list of identifiers separated by commas or it can be empty.

	Note: The first parentheses of the argument list must not be separated from the macro name by a whitespace, otherwise the macro will be treated as a simple macro.
--	---

The reference to the function macro should also be done in a form of a function call. The macro call arguments will be passed to the macro and processed by it.

Line No.	Processed Source file "a.4gl"	Line No.	Pre-Processor Output File
01	\$define	01	\$LINE 1 "a.4gl"
	func_macro(x,y) x+y		
02	\$define simple_macro	02	
	(x,y) x+y		
03	DISPLAY	03	
	func_macro(5,6)		
04	DISPLAY simple_macro	04	DISPLAY 5+6
	(3,4)		
05		05	DISPLAY (x,y) x+y
			(3,4)

If the argument list is empty, no parameters will be used by the macro, by the parentheses will still be necessary for the macro identifier to be recognized in the source code by the pre-processor. So if the macro



identifier is "macro()", only corresponding "macro()" identifiers will be expanded, whereas "macro" or "macro ()" identifiers will be skipped.

Macro Arguments

A function macro is processed in the following order:

1. All the arguments are expanded and substituted.
2. The macro itself is expanded.

The number of parameters should be the same for the macro definition and for the macro call. If the macro definition accepts only one parameter, a single parameter should be passed by the macro call. The macro call parameters can be literal values, variables or valid 4GL expressions separated by comma. If such an expression contains a comma symbol in itself, such expression should be taken into parentheses.

A macro argument can be left empty, if the macro call does not supply the values for the arguments.

However, if a macro has more than one argument, the argument list of the macro call should contain the corresponding number of comma delimiters, e.g.:

```
$define my_macro(a,b)  
my_macro(,) -- expanded normally  
my_macro(,2) -- expanded normally  
my_macro() -- causes an error  
my_macro(,,) -- causes an error
```

If you need to use NULL values use the NULL keyword as a parameter.

Using strings as parameters

For the parameter to be treated like a string instead of being treated like a 4GL expression, use the hash symbol (#) preceding the parameter in the macro body. When the macro parameter is preceded by (#) symbol, it is replaced by the literal string contained in the argument. The argument is not macro expanded before the substitution.

To concatenate tokens use the double hash mark (##). The tokens located before and after thus symbol will be concatenated.

In the example below the x argument is used as a 4GL expression returning TRUE in first and then as a character string later in the DISPLAY statement. If it had been used in the DISPLAY statement without the hash symbol, the result would have been: "1 is true".

Lin	Processed Source file "a.4gl"	Lin	Pre-Processor Output File
e		e	
No.		No.	

```
$define str(x) IF x THEN \  
DISPLAY #x||" is  
true" \  
END IF  
$LINE 1 "a.4gl"
```



```
$define concat(y) y, y ##  
_command  
str(1=1)  
  
concat(quit) IF 1=1 THEN DISPLAY  
"1=1"||" is true" END IF  
"quit", quit_command
```

Releasing Macros

You can release the macro name and then use the name in another \$define directive by using the following directive:

```
$undef name
```

Here, name is the name of a macro which you want to release. Since the pre processor executes files line by line, the macro released in such a way will have the effect above \$undef and will not be recognized below this directive.

Lin e No.	Processed Source file "a.4gl"	Lin e No.	Pre-Processor Output File
	\$define name "Hello"		\$LINE 1 "a.4gl"
	DISPLAY name		
	\$undef name		DISPLAY "Hello"
	DISPLAY name		
			DISPLAY name

Conditional Compilation

The pre-processor offers the functionality which allows the code to be compiled conditionally. A set of the following directives is used to compile the code conditionally:

```
$ifdef name  
...  
[$else  
...  
]
```



```
]  
$endif
```

If the name was defined previously using \$define directive, the code included after the directive name will be execute and the code in the \$else directive will not be executed. If the name was not defined, \$else directive will be executed instead.

You can use \$ifndef directive that has the same syntax and which condition is considered true, if the macro name was not defined. If it was defined, \$else directive will be executed instead.

```
$ifndef name  
...  
[$else  
...  
]  
$endif
```



Note: Even if the condition is not met and the code in \$ifdef or \$else directive is not executed, it is still checked for validity and tokenized. So the content of the \$ifdef and \$ifndef directives should be grammatically valid.

Lin e No.	Processed Source file "a.4gl"	Lin e No.	Pre-Processor Output File
	\$define name_defined		\$LINE 1 "a.4gl"
	\$ifdef name_defined		
	DISPLAY "TRUE - defined"		
	\$else		DISPLAY "TRUE - defined"
	DISPLAY "FALSE - undefined"		
	\$endif		
	\$ifndef name_defined		
	DISPLAY "TRUE - undefined"		
	\$else		



```
DISPLAY "FALSE - defined"
```

```
$endif
```

```
DISPLAY "FALSE - defined"
```

Predefined Macros

The pre-compiler offers two predefined variables that are extended when the source code is processed. These are:

- `__FILE__` - is expanded to a string constant specifying the name and the path (if specified) of the currently processed file. E.g. if the file was addressed by the following &include macro: "&include \"..//foo/source.4gl\"", the `__FILE__` macro will be expanded to ""..//foo/source.4gl".
- `__LINE__` - is expanded to the current line number of the source file being processed.



APPENDIX A: 4GL KEYS

THIS SECTION WILL LIST AND DESCRIBE ALL THE KEYS AND KEY COMBINATION WHICH CAN BE REFERENCED BY A 4GL PROGRAM OR USED DURING THE PROGRAM EXECUTION FOR NAVIGATION AND OTHER PURPOSES.



4GL Keys

This appendix contains the lists of the keys which can be referenced from within 4GL code or pressed to some effect during the program execution.

Context Actions

Different environments support different sets of keys. The tables below describe the keys available in different situations.

The Action column describes the effect which the key press causes.

The Logical Key column gives the name by which the key can be referenced from the 4GL code, for example in the OPTIONS ...KEY statement for the purpose of reassigning another keyboard key to the action. If this column is empty for a particular action, the action cannot be assigned a different key.

The Default column gives the name of the keyboard key (or key combination) which causes the action specified in the Action column. Key combinations are given with hyphens and mean that all the keys joined by hyphens must be pressed simultaneously.

Ring Menu or Toolbar

Action	Logical Key	Default	Comments
Navigate Previous Option		Cursor Up / Cursor Left	
Navigate Next Option		Cursor Down / Cursor Right / Space	
Select Current Option		Enter	

Editing General

Action	Logical Key	Default	Comments
Complete Input	ACCEPT	Escape	
Show Context Help	HELP	Control-W	
Navigate Next Field		Tab / Cursor Down ¹	
Navigate Previous Field		Shift-Tab / Cursor Up ¹	
Navigate Field Below		Cursor Down ²	
Navigate Field Above		Cursor Up ²	
Delete Character		Control-X	Character Mode Only
Clear to EOL		Control-D	Character Mode Only
Toggle Insert/Overwrite		Control-A	Character Mode Only
Move Cursor Right		Control-L	Character Mode Only
Move Cursor Left		Control-H	Character Mode Only
Screen Refresh		Control-R	Character Mode Only

¹ Constrained mode only

² Unconstrained mode only



Editing Arrays

Action	Logical Key	Default	Comments
Navigate Previous Row		Cursor Up	
Navigate Next Row		Cursor Down	
Navigate Previous Page	PREVIOUS	F3	
Navigate Next Page	NEXT	F4	
Insert New Row	INSERT	F1	
Delete Current Row	DELETE	F2	

Displaying Arrays

Action	Logical Key	Default	Comments
Navigate Previous Row		Cursor Up / Cursor Left	
Navigate Next Row		Cursor Down / Cursor Right / Enter	
Navigate Previous Page	PREVIOUS	F3	
Navigate Next Page	NEXT	F4	

Editing (Windows Graphical Environments)

Action	Logical Key	Default	Comments
Copy		Control-C	Depends on locale
Paste		Control-V	Depends on locale
Cut		Control-X	Depends on locale
Highlight character		Shift-Left/Right	
Highlight word		Control-Shift-Left/Right	
Highlight to end of field		Shift-End	
Highlight to start of field		Shift-Home	
Move Cursor Right		Cursor Right	
Move Cursor Left		Cursor Left	
Move Cursor Start of Word		Control-Cursor Left	
Move Cursor End of Word		Control-Cursor Right	
Move Cursor Start of Field		Control-Home	
Move Cursor End of Field		Control-End	

Miscellaneous

Action	Logical Key	Default	Comments
Interrupt Signal	INTERRUPT	Depends on stty settings (usually control-c). Emulated as control-c on windows/character.	Character only
Quit Signal		Depends on stty settings (usually control-z). Emulated as control-z on windows/character.	Character only
Interrupt Signal	INTERRUPT	Break	GUI clients only



Logical 4GL Keys

The table below gives the list of logical keys names that can be referenced from 4GL code (e.g. in the OPTIONS ... KEY statement).

The Action column describes the action invoked by the keypress, unless other actions are specified by means of an ON KEY logical block.

The Default column gives the default keyboard key or key combination bound with the logical key.

Key	Action	Default
ACCEPT	Complete Input	Escape
HELP	Show Context Help	Control-W
NEXT or NEXTPAGE	Scroll to Next Page (arrays only)	F4
PREVIOUS or PREVPAGE	Scroll to Previous Page (arrays only)	F3
INSERT	Insert new row (input array only)	F1
DELETE	Delete current row (input array only)	F2

Keyboard Keys

This table describes the way of referencing keyboard keys from the 4GL code.

The Key column contains the keyboard keys list.

Identifier column contains the way to reference the keyboard keys from the 4GL code.

The Comments column contains some additional information.

Key	Identifier	Comments
Enter	RETURN / "control-m"	
Tab	TAB / "control-i"	
a – z, A – Z, 0 - 9	a – z, A – Z, 0 - 9	
F1 – F1024	F1 – F1024 / "f1" – "f1024"	Keys F13 – F24 are triggered as Shift-F1 – Shift-F12 in most scenarios. For character mode, this depends entirely on your terminfo settings. Keys F25+ are virtual and cannot be triggered through the keyboard - they need to be assigned to form widgets.
Home	"key_home"	
End	"key_end"	
Delete	"delete"	
Backspace	"backspace"	
Break	"break"	



Cursor Up	UP / "up"	
Cursor Down	DOWN / "down"	
Cursor Left	LEFT / "left"	
Cursor Right	RIGHT / "right"	
Keypad Upper-Left	"key_a1"	
Keypad Upper-Right	"key_a3"	
Keypad Center	"key_b2"	
Keypad Lower-Left	"key_c1"	
Keypad Lower-Right	"key_c3"	
Keypad 0	"kp_0"	Requires 'num-lock'. Keypad keys override their standard counterparts if specified, otherwise the keypad keys trigger their standard counterparts.
Keypad 1	"kp_1"	
Keypad 2	"kp_2"	
Keypad 3	"kp_3"	
Keypad 4	"kp_4"	
Keypad 5	"kp_5"	
Keypad 6	"kp_6"	
Keypad 7	"kp_7"	e.g.
Keypad 8	"kp_8"	
Keypad 9	"kp_9"	ON KEY (0)
Keypad +	"kp_add"	# triggered by both keypad '0' and '0'
Keypad -	"kp_subtract"	
Keypad *	"kp_multiply"	
Keypad /	"kp_divide"	ON KEY ("kp_0")
Keypad .	"kp_decimal"	# only triggered by keypad 0.
Lower-Left	"key_ll"	
Insert Line	"key_il"	
Delete Line	"key_dl"	
Insert Character	"key_ic"	
Delete Character	"key_dc"	
Back-Tab	"key_btab"	
Begin	"key_begin"	
Cancel	"key_cancel"	
Clear	"key_clear"	
Clear All Tabs	"key_catab"	
Clear Tab	"key_ctab"	
Close	"key_close"	
Command	"key_command"	
Copy	"key_copy"	
Create	"key_create"	
Clear to End of Line	"key_eol"	
Clear to End of Screen	"key_eos"	
Enter	"key_enter"	
Exit	"key_exit"	
Find	"key_find"	
Help	"key_help"	
Mark	"key_mark"	
Message	"key_message"	
Move	"key_move"	
Open	"key_open"	
Options	"key_options"	
Previous	"key_previous"	



Print	"key_print"
Redo	"key_redo"
Reference	"key_reference"
Refresh	"key_refresh"
Replace	"key_replace"
Reset	"key_reset"
Restart	"key_restart"
Resume	"key_resume"
Save	"key_save"
Scroll Backward	"key_sr"
Scroll Forward	"key_sf"
Select	"key_select"
Send	"key_send"
Set Tab	"key_stab"
Suspend	"key_suspend"
Undo	"key_undo"
Shift-Begin	"key_sbegin"
Shift-Cancel	"key_scancel"
Shift-Command	"key_scommand"
Shift-Copy	"key_scopy"
Shift-Create	"key_screate"
Shift-Delete-Character	"key_sdc"
Shift-Delete-Line	"key_sdl"
Shift-Down	"key_sdown"
Shift-EOL	"key_seol"
Shift-Exit	"key_sexit"
Shift-Find	"key_sfind"
Shift-Home	"key_shome"
Shift-Insert-Character	"key_sic"
Shift-Insert-Line	"key_sil"
Shift-Left	"key_sleft"
Shift-Message	"key_smessage"
Shift-Move	"key_smove"
Shift-Next	"key_snext"
Shift-Options	"key_soptions"
Shift-Previous	"key_sprevious"
Shift-Print	"key_sprint"
Shift-Redo	"key_sredo"
Shift-Replace	"key_sreplace"
Shift-Right	"key_sright"
Shift-Resume	"key_srsume"
Shift-Save	"key_ssav"
Shift-Suspend	"key_ssuspend"
Shift-Undo	"key_sundo"



APPENDIX B: CLASSES

THIS SECTION WILL LIST AND DESCRIBE ALL THE CLASSES WITH SPECIAL MEANINGS WHICH CAN BE USED FOR FILTERING WIDGETS WITHIN A 4GL APPLICATION



Classes

This section contains the list and description of the classes with special meanings which can be used for filtering widgets within a 4GL application. The names of these classes fully correspond to the values added to the ClassNames property of a corresponding widget at runtime and can be used during the process of an application development.

For easy object manipulating using classes as filters, Lycia Theme Designer tool is strongly recommended to be used.

Depending on a **display attribute** specified in a source code, the following classes exist:

- attribute_dim
- attribute_bold
- attribute_invisible
- attribute_normal
- attribute_reverse
- attribute_underline
- attribute_black
- attribute_blue
- attribute_cyan
- attribute_green
- attribute_magenta
- attribute_red
- attribute_white
- attribute_yellow

These classes are automatically assigned to the ClassName property of a widget with a corresponding display attribute specified in an ATTRIBUTE clause on an application launch.

Each of the class name listed above corresponds to one of the display attributes available in Lycia. For example:

```
attribute "DIM"      =      class "attribute_dim ",  
attribute "BOLD"     =      class "attribute_bold",
```

and so on for INVISIBLE, NORMAL, REVERSE and UNDERLINE (font intencity and video); BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE and YELLOW (colour) attributes.

Usage:

The execution of the DISPLAY statement below will result in 'attribute_green', 'attribute_underline' and 'attribute_reverse' assignment to the ClassNames property of the Label:

```
DISPLAY "We have used the attributes: GREEN, UNDERLINE and REVERSE" AT 5,5  
ATTRIBUTE (GREEN, UNDERLINE, REVERSE)
```



Depending on a **front-end type** an application is run with, the following classes are available:

- lycia_normal a browser or a desktop client, but not a mobile device
- lycia_web a browser
- lycia_desktop LyciaDesktop
- lycia_touch a standalone mobile application
- lycia_mobile on a mobile device (a browser or as a standalone application)
- lycia_html an HTML client, that includes all three: LyciaWeb, LyciaDesktop, LyciaTouch

Depending on a **browser type** an application is run with, the following classes are available:

- lycia_firefox Mozilla Firefox
- lycia_opera Opera
- lycia_safari Safari
- lycia_ie Windows Internet Explorer

Depending on a **mobile OS** an application is run on, the following classes are available:

- lycia_android Android
- lycia_iphone iOS

Another classes with different meanings are as follows:

- full_screen enables the full-screen mode
- lycia_ActionsToolBar for action toolbar
- position_restore keeps the window/MDI frame size and position
- trusted enables a site inner page display (for the Browser widget)

By default, the browser widget does not allow a site inner page display with the ability to run scripts and submit forms, as this may cause the information about sensitive data in an application loss. To enable this ability back, the TRUSTED class name to the ClassName property of the browser widget must be added.

On doing this the security restrictions will be disabled, provided that there is no prohibition for opening the page in IFRAME tag on a site itself. Under this condition, it can only be opened via additional reverse proxy because the widget uses the IFRAME tag for running a page and does not depend on the browser version installed on the machine.

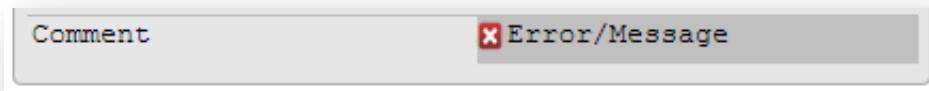
- queryEditable allows a COMBOBOX field editing provided that a CONSTRUCT statement is executed

For more detailed information regarding full_screen class and some other classes, that are normally used for the mobile app development, please, refer to the "Lycia Mobile" guide included to the package.



To manipulate the **StatusBar** widget appearance the following classes can be used:

- default



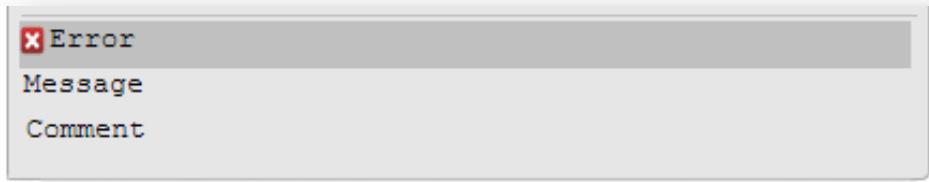
- lines1



- lines2



- lines3



- lines4

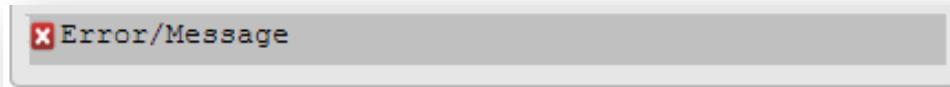




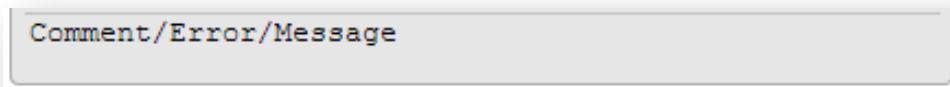
- lines5



- lines6



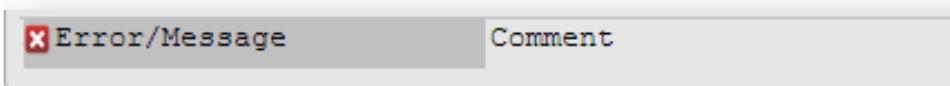
- panels1



- panels2



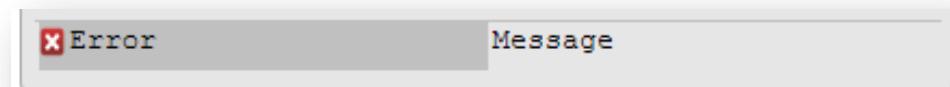
- panels3



- panels4

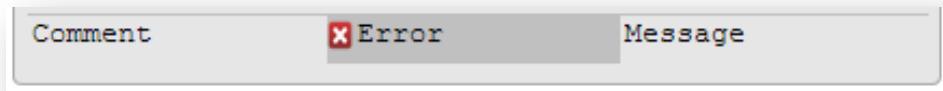


- panels5

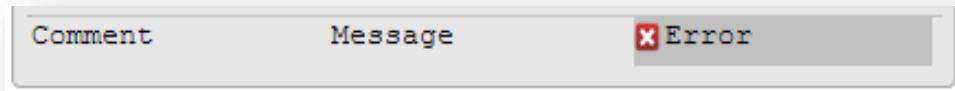




- panels6



- panels7



- none No Comment, Message or Error displayed



Index

4GL expressions	
<i>4GL vs. SQL</i>	74
<i>operands used in expressions</i>	79
<i>operators used in expressions</i>	75, 86, 88, 91, 94, 97
<i>parentheses used in expressions</i>	75
4GL reports	
<i>report destination</i>	404, 442
<i>report driver</i>	410, 436
<i>report features</i>	435
<i>report program block</i>	386, 437
<i>report prototype</i>	385
Asterisk (*) and THRU/THROUGH Keyword	102
CALL	114
CASE	118
CLEAR	122
<i>CLEAR FORM</i>	122
<i>CLEAR SCREEN</i>	122
<i>CLEAR WINDOW</i>	122
<i>field list</i>	123
CLOSE	
<i>CLOSE DATABASE</i>	124
<i>CLOSE FORM</i>	126
<i>CLOSE WINDOW</i>	127
CLOSE FILE	125
common clauses	
<i>ATTRIBUTE clause</i>	105, 139, 202, 282, 346, 373, 376
<i>declaration clause</i>	119
<i>field clause</i>	98
<i>HELP clause</i>	139, 203, 263, 285, 323, 375
<i>input control clause</i>	140, 264, 285, 377
<i>logical sub-blocks</i>	111, 230, 235, 252, 430
<i>RETURNING clause</i>	116, 394
CONSTRUCT	135
CONTINUE	153
CURRENT WINDOW	155
data types	
<i>classification</i>	9
<i>large</i>	14
<i>object</i>	13
<i>simple data types</i>	10
<i>structured</i>	13
data types conversion	
<i>characters and numbers</i>	68
<i>DATE and DATETIME</i>	67
<i>DATETIME and INTERVAL</i>	67
<i>large data types</i>	68
<i>number to number</i>	66
<i>simple data types conversion</i>	69
DATABASE	157, 319
<i>current database</i>	159
<i>default database</i>	158
DEFER	161, 319
<i>alternative for the DEFER statement</i>	424
DEFINE	163
<i>declaring a data type</i>	168
<i>declaring a variable</i>	164
<i>declaring variables</i>	318
DISPLAY	193
<i>displaying to console</i>	193
<i>displaying to form fields</i>	196
<i>displaying to screen or window</i>	194
DISPLAY ARRAY	201
DISPLAY FORM	219
END	221
ERROR	223
EXIT	225
FINISH REPORT	228
FOR	229
FOREACH	232
FUNCTION	239
<i>function body</i>	240
<i>web-function</i>	243
GLOBALS	245
GOTO	249
IF 251	
INITIALIZE	257
INPUT	259
INPUT ARRAY	279
LABEL	305
large data types	
<i>BYTE</i>	9, 19, 71
<i>TEXT</i>	10, 60, 72
LET	306
literals	
<i>DATETIME literal</i>	29, 94
<i>INTERVAL literal</i>	46, 94
<i>literal integer</i>	87
<i>literal numbers</i>	89
LOAD	309
LOCATE	314
MAIN	318
MENU	320
MESSAGE	334
NEED	336, 462
non-4GL data types	
<i>INTEGER8</i>	71
<i>NCHAR</i>	72
<i>NVARCHAR</i>	72
<i>SERIAL</i>	72
<i>SERIAL8</i>	72
numeric values	
<i>DATE numeric</i>	26, 93
<i>DATETIME numeric</i>	31, 94
<i>INTERVAL numeric</i>	48, 94
object data types	
<i>CURSOR</i>	9, 25, 71



FORM	10, 41, 71	CHAR/CHARACTER	9, 23, 71
PREPARED	10, 52, 72	character data types	12
WEBSERVICE.....	10, 64, 72	DATE.....	9, 26, 71
WINDOW.....	10, 65, 72	DATETIME	9, 27, 71
OPEN		DEC/DECIMAL.....	9, 33, 34, 71
OPEN FORM	340	FLOAT/DDOUBLE PRECISION	10, 40, 71
OPEN WINDOW.....	343	INT/INTEGER	10, 43, 71
OPEN FILE	337	INTERVAL.....	10, 44, 71
OPTIONS	351	MONEY.....	10, 50, 72
OUTPUT TO REPORT	362	numeric data types.....	11
PAUSE.....	364, 463	SMALLFLOAT/REAL.....	10, 57, 72
PREPARE	365	SMALLINT	10, 58, 72
PRINT.....	372, 465	STRING	10, 59, 72
PROMPT	374	time data types	13
qualifiers		TINYINT	10, 62, 72
DATETIME qualifier	28	VARCHAR.....	10, 63, 72
INTERVAL qualifier.....	44	SKIP	399, 475
table qualifiers.....	100	SLEEP.....	400
READ	383	SQL...END SQL block	401
REPORT.....	385	START REPORT.....	404
report sections		structured data types	
DEFINE section	440	ARRAY	9, 16, 71
FORMAT section	452	DYNAMIC ARRAY.....	10, 36, 71
ORDER BY section.....	449	RECORD	10, 54, 72
OUTPUT section.....	442	TERMINATE REPORT	412
reserved lines	220, 348, 353	TRY...CATCH	413
comment line.....	349	types of 4GL expressions	
error line.....	223	boolean expressions	82, 121
form line.....	220	character expressions.....	90
menu line.....	321	integer expressions.....	85
message line.....	334	numeric expressions.....	74, 88
prompt line.....	381	time expressions.....	92
RETURN	241, 389	UNLOAD	416
RUN	392	VALIDATE	420
SEEK.....	396	WHENEVER.....	421
simple data types		WHILE	430
BIGINT	9, 18, 71	WRITE	432
BOOLEAN.....	9, 21, 71		