NAME

assignment −assigning values to fields or variables

SYNTAX

Field Assignment:

dbfield-expression-1 ::= { *value-expression* | **edit** [*dbfield-expression-2*] }

dbfield-expression ::= [*context-variable*.]*field-name*

Variable Assignment:

variable-name = *value-expression*

DESCRIPTION

The *assignment* statement assigns values to fields in the **modify** and **store** commands, or values to variables. Chapter 5 of this manual contains a detailed discussion of the *assignment* statement.

ARGUMENTS

dbfield-expression-1 Specifies the field to receive a value. The *context-variable* optionally names the relation.

value-expression Specifies the value you want to assign to *dbfield-expression*. This value can be a quoted literal, a reference to another field, a prompting value expression (*.*'your own prompt'*), an aggregate or statistical expression, or the word **null** to assign the missing value.

dbfield-expression-2 Specifies the blob field whose contents you would like to edit for assignment to *dbfield-expression-1*.

variable-name Identifies the variable to which you want to assign a value. The variable must have been declared with a **declare variable** command.

value-expression Specifies the value you want to assign to *variable-name*.

EXAMPLE

The following example stores a record, using a **begin-end** statement to structure a compound statement for assigning values to each field:

```
QLI> store ski_areas using
CON>   begin
CON>     city = "Andover"
CON>     state = "MA"
CON>     name = "Parker State Forest"
CON>     type = "N"
```

```
        CON>  end
        QLI> The following example modifies a field value:

QLI> for c in cities with c.city = "Boston"
        CON>  modify using c.population = c.population * 1.10
        QLI>
```

SEE ALSO

Chapter 5 discusses the *assignment* statement in detail. See also the manual pages for **begin-end**, **modify**, and **store** statements in this manual.

DIAGNOSTICS

You may encounter the following messages when you use the *assignment* statement:

- *Operation failed on database "database-file-name"* with any of the following secondary messages:
  - *arithmetic exception, numeric overflow, or string truncation*. A value that you tried to store or modify did not fit. Check the field's characteristics and try again.
  - *attempt to store a duplicate value in a unique index*. A field value that you tried to store or modify violated the DUPLICATES NOT ALLOWED restriction for an index that includes that field. Try another value.
  - *conversion error*. This is a generic data conversion error that covers all but the following two cases.
  - *conversion error from string "out-of-range-date"* You tried to store or modify a date field with a value outside the range 1 January 100 to 11 December 5941, or an invalid date such as 29 February 1986. Try a value within the valid range. If the range is not adequate, you cannot use the date datatype.
  - *conversion to blob not supported*. You tried to store non-blob data in a blob field. Use the **edit** option described above.
  - *validation error for field field-name, value "supplied-value"*. A field value that you tried to store or modify violated the VALID_IF clause for a field. Check the valid values and try again.
- *"string" is undefined or used out of context*. This is a **qli** message in response to an unrecognized string.
- *execution terminated by signal*. You probably issued an end-of-file command.
- *user aborted (WC -Q) edit operation (display manager/Pad manager)*. You probably exited from the editor during a blob assignment without doing anything. This informational message comes from outside **qli**.
- *no permission for "type" access to "object"*. A security class exists for the specified object, and its access control list prohibits you from reading or writing that object.

- *action cancelled by trigger to preserve data integrity*. A trigger exists for an object you tried to modify or delete, and the corollary actions associated with the trigger prohibit that operation.

See also the discussion of errors in Chapter 1.

NAME

begin-end −begin/end block

SYNTAX

begin
qli-statement...
end

DESCRIPTION

The **begin-end** statement groups **qli** statements into a single compound statement called a *begin-end block*.

You can include in the begin-end block a procedure that specifies some complete operation. For example, you may have a procedure that prompts you for values and updates a record. You can use fragmentary procedures, but you must include enough additional information to make complete statements.

ARGUMENTS

qli-statement Any of the **qli** statements listed in Chapter 1, and the **repeat** command.

EXAMPLE

The following example stores a record, using a begin/end block for assignment statements:

```
QLI> store cities using
CON>   begin
CON>     city = 'Shadkill'
CON>     state = 'NY'
CON>     population = 20000
CON>     altitude = 17
CON>   end
QLI>
```

SEE ALSO

The manual pages for the statements listed in *Syntax*.

DIAGNOSTICS

You may encounter the following message when you use the **begin-end** statement:

- *expected statement, encountered "command".*

  You included a command in the begin/end block. Use only the statements listed in the *Syntax* section above.

See also the discussion of errors in Chapter 1.

NAME

boolean-expression −relationship between value expressions

SYNTAX

boolean-expression ::= { [**not**] *conditional-expression* |
conditional-expression **and** *conditional-expression* |
conditional-expression **or** *conditional-expression* }

conditional-expression ::= { *comparison-condition* | *between-condition* |
starting-condition | *containing-condition* |
matching-condition | *not-condition* | *unique-condition* }

DESCRIPTION

A *boolean-expression* evaluates to true, false, or missing. It describes the characteristics of a single value expression (for example, a missing value) or the relationship between two value expressions (for example, *x* is greater than *y*).

The order of precedence for evaluating compound Boolean expressions is **not**, **and**, and **or**.

ARGUMENTS

comparison-condition Describes the characteristics of a single expression. The format of the *comparison-condition* follows:

Syntax: comparison-condition of Boolean Expression

value-expression-1 relational-operator value-expression-2

The *relational-operator* can be any of the operators in the following table:

Operator	Relationship
eq *or* = *or* ==	equal
ne *or* <> *or* !=	not equal
gt *or* >	greater than
ge *or* >=	greater than or equal
lt *or* <	less than
le *or* <=	less than or equal

between-condition Tests whether a value expression, *value-expression-1*, occurs between two other value expressions, *value-expression-2* and *value-expression-3*. This test is inclusive of the boundary values. The format of the *between-condition* follows:

Syntax: between-condition of Boolean Expression

value-expression-1 [**not**] **between**
value-expression-2 **and** *value-expression-3*

containing-condition Tests for the presence of *string* (case-*in*sensitive) anywhere in *value-expression*. It evaluates to true if *string* is contained in *value-expression*. If the value of *value-expression* is missing, the result is missing. The format of the *containing-condition* follows:

Syntax: containing-condition of Boolean Expression

value-expression-1 [**not**] **containing** *value-expression-2*

Qli recognizes **ct** and **cont** as synonyms for **containing**. *starting-condition* Tests for the presence of *string* (case-sensitive) at the beginning of *value-expression*. It evaluates to true if the first characters of *value-*

expression match *string*. The search is case-sensitive. The format of the *starting-condition* follows:

Syntax: Starting Boolean Expression

value-expression-1 [**not**] **starting with** *value-expression-2*

matching-condition Tests for the presence of *wildcarded-string*, a string that can contain the wildcard characters ***** and **?**. The asterisk matches an unspecified run of characters, while the question mark matches a single character. This test is case *in*sensitive. The format of the *matching-condition* follows:

Syntax: matching-condition of Boolean Expression

value-expression-1 [**not**] **matching** *value-expression-2*

missing-condition Tests for the absence of a value in *dbfield-expression*. It is true if the value of *dbfield-expression* is missing. The format of the *missing-condition* follows:

Syntax: missing-condition of Boolean Expression

dbfield-expression [**not**] **missing**

Unless you specify otherwise in the field's definition, **qli** prints blanks for numbers, characters, and dates, and nothing for blobs. See the for more information about defining alternate missing values.

any-condition Tests for the existence of at least one qualifying record in a relation or relations. This expression is true if the record stream specified by *rse* includes at least one record. If you add **not**, the expression is true if there are *no* records in the record stream. The format of the *any-condition* follows:

Syntax: any-condition of Boolean Expression

[**not**] **any** *rse*

You might want to use **any** instead of joining records if all you want to do is establish that a record exists. As soon finds one record that meets the search criteria, it stops, whereas a join would continue until it found all qualifying records.

unique-condition Tests for the existence of exactly one qualifying record. This expression is true if the record stream specified by *rse* consists of only one record. If you add **not**, the condition is true if there is more than one record in the record stream or if the record stream is empty. The format of the *unique-condition* follows:

Syntax: unique-condition of Boolean Expression

[**not**] **unique** *rse*

EXAMPLES

The following query looks for cities with populations between 100,000 and 250,000:

```
QLI> for cities with population between 100000 and 250000
CON> print city, state, population
```

The following query looks for cities with the substring *ville* somewhere in their name:

```
QLI> print cities with city containing 'ville'
```

The following query looks for cities that start with the string *New*:

```
QLI> print states with state_name starting with 'New'
```

The following query looks for cities with the string *ton* following any number of other characters:

```
QLI> print cities with city matching '*ton*'
```

The following query looks for states with the state abbreviation equal to *N* followed by exactly one character:

```
QLI> print states with state matching 'N?'
```

The following query looks for states that have a missing value for the CAPITOL field:

```
QLI> print states with capitol missing
```

The following query prints the name of any state for which there are cities stored:

```
QLI> for s in states with any c in cities over state
CON> print s.state_name
```

The following query prints the names of states that have only one ski area:

```
QLI> for s in states with unique ski in ski_areas over state
CON> print s.state_name
```

SEE ALSO

value-expression (qli), *rse* (qli)

NAME

commit −write changes to database

SYNTAX

commit [*database-handle-commalist*]

DESCRIPTION

The **commit** command writes to the database changes made during the transaction.

You can use the **commit** statement in conjunction with the **prepare** statement to execute a two-phase commit. The database software automatically executes such a commit when necessary, but ,if required, you can control the two-phase commit explicitly. See the manual page for **prepare** in this manual.

ARGUMENTS

database-handle Specifies a name that can be used to qualify database reference when you are using multiple databases. If you do not specify a database handle, the **commit** command affects all open databases; writes to the database(s) all changes to data and metadata. It also flushes all modified buffers and closes any record streams that are open.

If you assign a database handle when you ready the database, you can use the handle to limit the effect of the **commit** to specific databases. When you access more than one database in **qli**, the database software automatically starts up separate subtransactions for each database. However, these behave as a single transaction. The optional *database-handle* lets you control these subtransactions explicitly by letting you commit or roll back transactions by database.

If you forgot to assign a database handle when you readied the database and run into a problem with a database while you have several open, do not despair; **qli** assigns a default handle if you have not specified one. Type the following to find out the default database handle assigned by **qli**:

```
QLI> show databases
        Database "atlas.gdb" readied as QLI_0
        QLI>
```

Qli displays the names of all available entities, including databases and handles. The default handles are of the form ‘‘QLI_n,’’ where *n* is a numeric integer. Supply this handle as an argument to the **commit** command:

```
QLI> commit qli_1
        QLI>
```

EXAMPLE

The following **qli** script readies a database, stores a record, thus starting a transaction, and then commits the transaction:

```
QLI> ready atlas.gdb
     QLI> store ski_areas
                    .
                    .
                    .
     QLI> commit
     QLI>
```

SEE ALSO

rollback (qli), **finish** (qli), **prepare** (qli)

DIAGNOSTICS

You may encounter the following message when you use the **commit** statement:

- *expected database handle, encountered <string>*.

  You need a database handle. You may have mistyped the handle. Use the **show databases** command to check the database handle.

See also the discussion of errors in Chapter 1.

NAME

declare variable −local and global variables

SYNTAX

declare *variable-name datatype*

datatype ::= { **short** [*scale-clause*] | **long** [*scale-clause*] | **float** | **double** | **char**[*n*] | **varying**[*n*] | **date** }

scale-clause ::= **scale** [-]*n*

DESCRIPTION

The **declare** statement lets you declare local and global variables for use in **qli**. ‘‘Local scoping’’ refers to use within a begin—end block. You can use variables in statements, reports, and procedures. To assign a value to a variable, use the *assignment* statement.

This table lists the datatypes by size and range/precision.

Datatype	**Size**	**Range/Precision**
short	16 bits	-32768 to 32767
long	32 bits	-2**31 to (2**31)-1
float	32 bits	approx. 7 decimal digits
double	64 bits	approx. 15 decimal digits
char[*n***]**	*n* bytes	0 to 32767 characters
varying[*n***]**	varies up to *n* bytes	0 to 32767 characters
date	64 bits	1 January 100 to 11 December 5941

ARGUMENTS

variable-name Names the variable. The variable name must start with an alphabetic character (*a*-—*z*), and can contain numbers, underscores, and dollar signs.

scale-clause Specifies the power of 10 by which the database software multiplies the stored integer value for use by **qli**. None of the supported languages supports the scale factor. For example, a negative scale of two means that there should be a decimal point two places to the left of the digits.

EXAMPLES

The following command declares a variable GLARP, assigns 617, and prints the variable's value:

```
QLI> declare glarp long
QLI> glarp = 617
QLI> print glarp
.sp 0.5
  GLARP
=========
      617
.sp 0.5
QLI> The following extract uses a prompting expression in the variable assignment:
```

```
QLI> declare glarp long
QLI> glarp = *'value for glarp'
Enter value for glarp: 412
QLI>
```

SEE ALSO

assignment (qli)

DIAGNOSTICS

You may encounter the following message when you use the **declare variable** statement:

- *expected field definition clause, encountered "bad string"*

  You tried to declare a variable, but the field definition was incorrect. Check the syntax and try again.

See also the discussion of errors in Chapter 1.

NAME

define procedure −storing procedure

SYNTAX

define procedure [*database-handle.*]*procedure-name*
operation...
end_procedure

operation ::= { *qli-command* | *qli-procedure* |
qli-statement | *qli-clause* | *qli-keyword* }

DESCRIPTION

The **define procedure** statement stores a sequence of **qli** operations in the database.

You can include a running commentary in your procedures so that other people can figure out what it does. Comment lines in **qli** begin with a slash and asterisk and end with an asterisk slash:

```
QLI> define procedure gl451
CON> /* This procedure does some useful things
CON> Well, maybe not really useful,
CON> but it does do something. */
CON> print states with capitol = *.'capitol city'
```

ARGUMENTS

[*database-handle.*]*procedure-name* Names the procedure. The procedure name can be up to 31 characters and can contain alphabetic characters (A—Z and a—z, all stored as uppercase), numeric characters (0—9), underscores (_), and dollar signs ($). The procedure name must start with an alphabetic character.

The optional database handle specifies the database in which the procedure is stored.

qli-statement Any of the **qli** statements listed in Chapter 1.

qli-command Any of the **qli** commands listed in Chapter 1.

qli-clause A clause from a **qli** command or statement.

qli-keyword A **qli** keyword.

EXAMPLE

The following sequence of commands defines a procedure that finds a record using a prompted-for value:

```
QLI> define procedure capitol_info
CON>   /* saves typing a common query */
CON>   for s in states cross c in cities over state with
CON>       s.state = *.'state code' and
CON>       s.capitol = c.city
CON>   print s.capitol | ' has a population of ' | c.population
CON> end_procedure
QLI> :capitol_info
Enter state code: AZ
Phoenix has a population of 789704
QLI>
```

SEE ALSO

See Chapter 8 of this manual for a complete discussion of procedures. See also the manual pages for **edit procedure**, **delete procedure**, and **rename procedure** in this manual.

DIAGNOSTICS

You may encounter the following message when you use the **define procedure** statement:

- *procedure name <name> in use.*

  Choose another name.

- *procedure name over 31 characters.*

  Choose a shorter name.

- *gds_$create_blob failed.*

  The database software could not create the field in which the procedure text is stored. Try again.

- You may get the following errors when you execute a procedure:

  — *procedure <name> is undefined.*

  The procedure does not exist as specified. Type *show procedures* for a list of procedures.

  — *Procedure <name> not found.*

  The procedure does not exist as specified. Type *show procedures* for a list of procedures.

See also the discussion of errors in Chapter 1.

NAME

delete procedure −deleting procedure

SYNTAX

delete procedure *procedure-name*

DESCRIPTION

The **delete procedure** command deletes a stored procedure.

ARGUMENTS

procedure-name Specifies the procedure you want to delete.

EXAMPLE

The following sequence of commands deletes a procedure:

```
QLI> delete procedure sunbelt_cities
QLI>
```

SEE ALSO

See Chapter 8 of this manual for a comprehensive discussion of procedures. See also the manual pages for **define procedure** and **edit procedure**.

DIAGNOSTICS

You may encounter the following message when you use the **delete procedure** statement:

- *procedure <name> not found*. The procedure does not exist as specified. Type *show procedures* for a list of procedures.

See also the discussion of errors in Chapter 1.

NAME

delete −erase record

SYNTAX

delete from *relation-name* [*alias*] [**where** *predicate*]

DESCRIPTION

The **delete** statement erases one or more records in a relation.

If you do not provide a search condition, deletes all records in *relation-name*. Be very careful with this option.

You cannot delete records from views or joins. Rather, you must erase them through the source relations.

ARGUMENTS

relation-name Specifies the relation from which a record is to be deleted.

alias Qualifies field references with an identifier that indicates the source relation. The *alias* can be useful if *predicate* references sources with overlapping field names.

The *alias* can contain up 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character (*A—Z*, *a—z*).

where *predicate* Determines the record(s) to be deleted. If you provide a search condition with the optional **where** *predicate* clause, deletes the record(s) selected from *relation-name*.

EXAMPLE

The following statement deletes all records from the CITIES relation with a value for POPULATION of less than 100,000:

```
QLI> delete from cities-
CON>   where population < 100000;
QLI>
```

The following statement deletes the entire TERRITORIES relation:

```
QLI> delete from territories;
QLI>
```

The following example deletes all qualifying records:

```
QLI> delete from ski_areas-
CON>    where name = 'Birchwood Slopes';
```

SEE ALSO

predicate (qli), *select-expression* (qli)

DIAGNOSTICS

You may encounter the following messages when you use the **delete** statement:

- *can't erase from a join*.

  You tried to erase from a join. This is an illegal operation. If you want to erase records in different relations, you must do so in separate statements.

- *no context for ERASE*.

  You did not provide a record selection expression. Try again.

See also the discussion of errors in Chapter 1.

NAME

edit −editing commands

SYNTAX

edit

DESCRIPTION

The **edit** command calls your default text editor, places the contents of your last **qli** command or statement in the editing buffer, and then deposits you in that buffer. You can then revise the **qli** command or statement.

Use the standard editing commands to change the command line as you want. You can also use this command to repeat the previous command by not changing the contents of the editing buffer, and then exiting. In either case, when you finish editing, exit from the editor as you normally do. **Qli** automatically executes the revised command.

EXAMPLE

The following example corrects a syntactic error:

```
QLI> print cities with state = New York
     ** QLI error: expected end of statement, encountered "YORK"
     QLI> edit
```

Qli calls your default editor. Add quotes around the string *New York*. Exit from the editor. **Qli** executes the query and displays the records.

```
QLI>
```

DIAGNOSTICS

The only errors you will receive from this command are those generated by your editor. Problems may include a lack of disk space or protection violations that prevent the editor from opening a scratch or journal file.

NAME

edit procedure −editing or creating procedures

SYNTAX

edit [*database-handle.*]*procedure-name*

DESCRIPTION

The **edit procedure** command lets you change a stored procedure or create a new one.

When you issue the **edit procedure** command, **qli** calls your default editor. If the procedure already exists, **qli** writes the text of the procedure to the editing buffer. The buffer does not include the **define procedure** and **end_procedure** structure. Use the standard editing commands to change the procedure as you want.

If the procedure does not exist, **qli** opens an empty edit window or buffer. Enter **qli** commands or statements. Do not use the **define procedure and end_procedure** commands that you would use to define a procedure at the `QLI>` prompt. **Qli** supplies these for you.

When you finish editing a procedure or inserting a new one, exit from the editor as you normally do. **Qli** automatically stores the procedure in the database.

ARGUMENTS

[*database-handle*.]*procedure-name* Names the procedure you want to edit or create.

The optional database handle specifies the database in which the procedure is stored.

EXAMPLE

The following example edits a procedure:

`QLI>` *edit high_cities*

Qli calls your default editor. Edit the procedure and exit.

SEE ALSO

See Chapter 8 of this manual for a comprehensive discussion of procedures. See also the manual pages for **define procedure** and **delete procedure**.

DIAGNOSTICS

You may encounter the following messages when you use the **edit procedure** command:

- *procedure name <name> in use*.

  Choose another name.

- *procedure name over 31 characters*. Choose a shorter name.

- *gds_$create_blob failed*.

The database software could not create the field in which the procedure text is stored. Try again.

- Errors from your editor. Possible problems include a lack of disk space or protection violations that prevent the editor from opening a scratch or journal file.

See also the discussion of errors in Chapter 1.

NAME

erase −erase record

SYNTAX

erase [[**all of**] *rse*]

DESCRIPTION

The **erase** statement removes from the database the record(s) specified by the record selection expression.

You cannot erase records from views or joins. Rather, you must erase them through the source relations.

ARGUMENTS

rse

all of *rse* Specifies that selected by the specified record selection expression are to be deleted:

- If you do not specify an *rse*, you must select the record or records to be deleted in an outer **for** loop.
- If you do specify an *rse*, you must provide a complete record selection expression in the **erase** statement.

EXAMPLE

The following example identifies the record to be deleted in an *rse* in the **erase** command itself:

```
QLI> erase all of cities with population < 100000 or
CON>   population missing
QLI> The following example uses a for loop to identify the record you want to erase:
```

```
QLI> for cities with population < 100000 or population missing
CON>     print then
CON>       if *.'keep it?' containing 'n' erase
QLI>
```

SEE ALSO

for (qli), *rse* (qli)

DIAGNOSTICS

You may encounter the following messages when you use the **erase** statement:

- *can't erase from a join.*

  You tried to erase from a join. This is an illegal operation. If you want to erase records in different relations, you must do so in separate statements.

- *no context for erase*.
  You did not provide a record selection expression. Try again.

See also the discussion of errors in Chapter 1.

NAME

exit −exiting qli

SYNTAX

exit

DESCRIPTION

The **exit** command ends a **qli** session and commits the current transaction.

Exit, **quit**, and the end-of-file character are exactly equivalent. The end-of-file characters are system-dependent:

- *Control-Z* for VAX/VMS, MicroVMS, and APOLLO
- *Control-D* for ULTRIX and SUN

EXAMPLE

```
QLI> exit
```

☞

SEE ALSO

quit (qli), **commit** (qli), **finish** (qli)

NAME

finish −close database

SYNTAX

finish [*database-handle-commalist*]

DESCRIPTION

The **finish** command explicitly closes a database.

If you close a database and want to access it later, you must ready it again.

ARGUMENTS

database-handle Specifies the database to close. If you do *not* specify a database handle, the **finish** command implicitly commits all default transactions and closes all open databases.

If you close a specific database, commits the default transaction for that database.

If you neglected to declare a database handle when you opened the database, you can use the default handle declared by **qli**. Use the **show databases** command to display the name of the handle that **qli** declared for the database.

EXAMPLE

The following command closes all open databases:

```
QLI> finish
```

The following example readies two databases, performs some data manipulation, and closes one of the databases:

```
QLI> ready /usr/igor/datafiles/atlas.gdb as atlas
    QLI> ready maps.gdb as map
        .
        .
        .
    QLI> finish atlas
```

SEE ALSO

ready (qli), **commit** (qli), **rollback** (qli)

DIAGNOSTICS

You may encounter the following message when you use the **finish** command:

- *expected database handle, encountered <string>*.

You need a database handle. You may have mistyped the handle. Type *show databases* to check the database handle.

See also the discussion of errors in Chapter 1.

NAME

for −repeating loop

SYNTAX

for *rse qli-statement*

DESCRIPTION

The **for** statement evaluates a record selection expression and executes a substatement for each qualifying record.

You can nest **for** loops to display a hierarchy of records or to join relations across databases.

ARGUMENTS

rse Provides the record selection criteria to form a record stream.

qli-statement Any of the **qli** statements listed in Chapter 1.

EXAMPLE

The following example creates a record stream **for** loop and displays records from that stream:

```
QLI> for states sorted by state
CON>   print capitol, state, statehood
```

The following example joins two relations:

```
QLI> for states cross cities over state sorted by city
CON>   print city, state, altitude, population
```

The following example uses a for loop to select records to be erased and then erases them:

```
QLI> for ski_areas with state = 'FL'
CON>   erase
```

The following example displays a hierarchy of records in a sort of part—component application. It picks up a value from an outer loop, prints it, and then prints associated values from another relation in an inner loop:

```
QLI> for r in rivers sorted by river
CON>   begin
CON>   print river
CON>   for rs in river_states with r.river = rs.river
```

```
CON>      print state
CON>   end
```

The following example is equivalent to a join operation across databases:

```
QLI> ready apollo:/usr/data/mapper.gdb as mapper
     QLI> ready atlas.gdb as atlas
     QLI> for s in atlas.states sorted by s.state
     CON> begin
     CON>   for c in mapper.cities with
     CON>      s.state = c.state
     CON>      print s.state_name, c.city, c.population
     CON> end
```

Note that you cannot reference relations from more than one database in a record selection expression. Therefore, **for** loops are the way to combine relations across databases.

SEE ALSO

rse (qli)

DIAGNOSTICS

You may encounter the following message when you use the **for** statement:

- *relations from multiple databases in single rse* and *can't mix databases within RSE*.

  You tried to access more than one database in the same record selection expression. Use nested **for** statements to do that.

See also the discussion of errors in Chapter 1.

NAME

help −online assistance

SYNTAX

help [*qli-command* | *qli-statement*]

DESCRIPTION

The **help** command provides assistance on **qli** commands and statements. If you do not ask for help on a *command* or *statement*, **qli** displays a listing of what help is available. If you ask for help on a subject for which there is no assistance, **qli** tells you that no help is available for that subject.

The first time you ask for help, there may be a slight delay as readies the database containing the help topics.

ARGUMENTS

qli-command Specifies the **qli** command for which you want help.

qli-statement Specifies the **qli** statement for which you want help.

MODIFYING HELP

You can edit the help files, add new topics, or delete existing ones. For example, you may want to replace frivolous examples with ones more closely tied to your application or document procedures that you want everyone to use. To do any of these things, invoke **qli** and ready the help database:

- *sys$help:help.gdb* on VAX/VMS systems
- */usr/gds/help/help.gdb* on UNIX systems
- */sys/gds/help.gdb* on DOMAIN systems

Use the **show** commands to see the record structure of the help database:

```
QLI> show fields
    Database "help.gdb"
        TOPICS
            TOPIC               text, length 31
            FACILITY            text, length 6
            SYSTEM_FLAG         text, length 1
            TEXT                blob, segment length 80
    .sp
    QLI>
```

The database *help.gdb* contains all the help topics. Now that you have readied the help database, you can manipulate it as you would your own database. To modify a topic, select the record with an RSE and change field values. For example:

```
QLI> modify text of topics with topic = STORE
```

> **Qli** calls your default editor. Make any changes you want to the text, and then exit from the editor in the normal manner.

```
QLI>
```

To store a new topic, use the **store** or **insert** statement. For example:

```
QLI> store topics
     Enter TOPIC: LJUBJANKA
     Enter SYSTEM_FLAG: X
     Enter FACILITY: NKVD
```

> **Qli** calls your default editor. Make any changes you want to the text, and then exit from the editor in the normal manner.

```
QLI>
```

Standard help topics have a system flag of ''S''; your messages should use some other flag value so they will not be overwritten when new software versions update the help library.

Finally, use the **erase** statement to remove unwanted topics:

```
QLI> for topics with topic = SELECT
     CON>  erase
     QLI>
```

As you enter new topics and modify existing ones, you should pay attention to the form of the TOPIC name. Whenever someone asks for help on a particular subject, the help facility in **qli** searches through TOPICS for a match on the TOPIC field. You have problems with multiple word topics unless you use underscores between them.

EXAMPLE

The following command displays the general help listing:

```
QLI> help
```

The following command displays help about the **store** statement:

```
QLI> help store
```

SEE ALSO

modify (qli), **store** (qli), **erase** (qli)

DIAGNOSTICS

You may encounter the following message when you use the **help** command:

- *No help is available for "subject".*
  The subject for which you requested help does not exist. Type *help* for a list of topics.

See also the discussion of errors in Chapter 1.

NAME

if—else −if—then—else construct

SYNTAX

if *boolean-expression* [**then**]
qli-statement
else
qli-statement

DESCRIPTION

The **if—else** statement provides an if—then—else structure in **qli**. The optional **then** in the syntax is a noiseword.

ARGUMENTS

qli-statement Any of the statements listed in Chapter 1 or a procedure. Generally speaking, it will be a **begin—end** statement. **Qli** requires a hyphen after the first **end**. For example:

```
if expression
    begin
      qli-statement
      qli-statement
    end -
  else
    begin
      qli-statement
      qli-statement
    end
```

boolean-expression Specifies a condition that must be true for the **if** to be executed. If the condition is not true, the **else** branch is executed.

EXAMPLE

The following fragment demonstrates an **if—else** in **qli**:

```
QLI> if full_name = " " or full_name missing
CON>   print ...
CON>   else print ...
```

See Chapter 7 for a complete example.

DIAGNOSTICS

You may encounter the following message when you use the **if—else** statement:

- *expected statement, encountered "command"*. You cannot use a command with the **if—else** statement.

See also the discussion of errors in Chapter 1.

NAME

insert −store a record

SYNTAX

insert into *relation-name* (*database-field-commalist*)
{ **values** *constant-commalist* | *select-statement* }

DESCRIPTION

The **insert** statement stores a new record into a relation.

You cannot use the SQL variant of **qli** to store records with blob fields. Use the **store** statement to store such records.

ARGUMENTS

relation-name Specifies the relation into which you want to store a new record.

database-field Lists the field in *relation-name* for which you are providing a value.

If you want to store the missing value for a field, do not reference that field in the **insert** statement.

If the database field is a blob, you can only assign the **null** value.

constant Provides a value for *database-field*. You can assign field values by inserting quoted strings, and quoted or unquoted numbers.

select-statement Specifies that the values for the new record are to come from the record identified by a **select** statement.

EXAMPLES

The following statement inserts quoted values:

```
QLI> insert into ski_areas (name, type, city, state)
CON>   values ('Radar Acres', 'N', 'Dunstable', 'MA');
```

The following example stores a new record into CITIES, using most of the values from an existing record:

```
QLI> insert into cities-
CON>   (city, state, latitude_degrees, latitude_minutes,
CON>   latitude_compass, longitude_degrees, longitude_minutes-
CON>   longitude_compass)
CON>   select 'Troy', state, latitude_degrees, latitude_minutes,
CON>   latitude_compass, longitude_degrees, longitude_minutes,
```

```
CON>  longitude_compass-
CON>  from cities where city = 'Albany' and state = 'NY'
```

The following statement stores a new record into CITIES, implicitly assigning the missing value to all unreferenced fields:

```
QLI> insert into cities-
CON>  (city, state)-
CON>  values ('Lowell', 'MA');
```

The following statement stores a new record into TOURISM, but does not reference the blob fields OFFICE or GUIDEBOOK, thereby assigning the missing value to those fields:

```
QLI> insert into tourism-
CON>  (state, zip, city)-
CON>  values ('NY', '10022', 'New York');
```

SEE ALSO

select (qli), **store** (qli)

DIAGNOSTICS

See the manual page for the *assignment* statement. See also the discussion of errors in Chapter 1.

NAME

list −displaying records

SYNTAX

Standalone format:

list *value-expression-commalist* **of** *rse*
[**on** *'filespec'* | **to** *shell-command*]

For loop format:

for *rse*
list *value-expression-commalist*

DESCRIPTION

The **list** statement displays fields from records in a record stream. Unlike the **print** statement, it displays the field values in a vertical format.

ARGUMENTS

value-expression **of** *rse* Specifies a list of fields or other values from the record stream created by the record selection expression.

on *'filespec'* Sends the output to the named, quoted file, rather than writing it to your monitor.

to *shell-command* Sends the output to standard input of the shell or command interpreter command, rather than writing it to your screen. These commands typically send the output to a printer, as in **print**, **lpr**, **lpt**, **prf**, and **'prf -npag'**. Note that if you include a switch on the shell command, you must quote the entire command.

EXAMPLE

The following query lists all records in STATES:

```
QLI> list states
```

QLI> The following query includes a **for** loop that selects records:

```
QLI> for states with area lt 10000
CON>     print state_name, area
```

The following query writes field values from STATES to the file *state_data.dat*:

```
QLI> list state, capitol, area of states on 'state_data.dat'
```

SEE ALSO

print (qli)

DIAGNOSTICS

You may encounter the following message when you use the **list** statement:

- *no items in print list*. You must provide a record selection expression or value expression.
- *Can't open output file*. **Qli** cannot open an output file for a *print on filespec* command.

See also the discussion of errors in Chapter 1.

NAME

modify −change field value

SYNTAX

modify { *dbfield-expression-commalist* | **using** *assignment-statement* } [**of** *rse*]

dbfield-expression ::= [*context-variable.*]*field-name*

DESCRIPTION

The **modify** statement updates a field or fields in a record or records.

ARGUMENTS

dbfield-expression Specifies the field you want to update. **Qli** prompts you for a field value.

assignment-statement Assigns the supplied constant or missing value to the field.

If you want to modify a blob field, use either the **edit** option of the *assignment* statement or **qli**'s prompting feature.

rse Specifies record selection criteria.

If you do not supply *rse*, you must enclose the **modify** command in a **for** loop that contains a record selection expression.

You cannot update records through a view. If you want to update such records, you must change them through the source relations.

EXAMPLE

The following statements change the same record using the **modify** statement in different ways:

```
QLI> /* rse in modify statement */
     QLI> modify population of cities with
     CON>   city = 'New York'
     Enter POPULATION: 10000000
     .sp
     QLI> /* rse in for statement */
     QLI> for cities with city = 'New York'
     CON>   modify using population = 10000000
     .sp
     QLI> modify population of cities with
     CON>   city = 'New York'
```

```
Enter POPULATION: 10000000
QLI>
```

SEE ALSO

assignment (qli), **for** (qli), **begin-end** (qli)

DIAGNOSTICS

See the manual page for the **assignment** statement. See also the discussion of errors in Chapter 1.

NAME

predicate −specify Boolean expression

SYNTAX

predicate ::= { *condition* | *condition* **and** *predicate* |
condition **or** *predicate* | **not** *predicate* }

condition ::= { *compare-condition* | *between-condition* |
like-condition | *in-condition* | *exists-condition* | *(predicate)* }

DESCRIPTION

The *predicate* clause is used to select the records to be affected by the statement. It is used in the *where-clause* of the **delete** and **update** statements and in the *select-expression*.

ARGUMENTS

compare-condition The *compare-condition* describes the characteristics of a single scalar expression (for example, a missing or null value) or the relationship between two scalar expressions (for example, *x* is greater than *y*).

Syntax: compare-condition of Predicate

{ *scalar-expression comparison-operator scalar-expression* |
scalar-expression comparison-operator (column-select-expression) |
scalar-expression **is** [**not**] **null** }

comparison-operator ::= { = | ^= | < | ^< | <= | > | ^> | >= }

column-select-expression ::=
select [**distinct**] *scalar-expression from-clause* [*where-clause*]

between-condition The *between-condition* specifies an inclusive range of values to match.

Format: between-condition of Predicate

database-field [**not**] **between** *scalar-expression-1*
and *scalar-expression-2*

like-condition Matches a string with the whole or part of a field value. The test is case-sensitive.

Format: like-condition of Predicate

database-field [**not**] **like** *scalar-expression*

The *scalar-expression* usually represents an alphabetic or numeric literal, and can contain wildcard characters. Wildcard characters are:

- The underscore, _, that matches a single character.
- The percent sign, %, that matches any sequence of characters, including none. You should begin and end wildcard searches with the percent sign so that you match leading or trailing blanks.

in-condition Lists a set of scalar expressions as possible values.

Format: in-condition of Predicate

scalar-expression [**not**] **in** (*set-of-scalars*)
set-of-scalars ::= { *constant-commalist* | *column-select-expression* }
column-select-expression ::= **select** [**distinct**] *scalar-expression from-clause* [*where-clause*]

exists-condition Tests for the existence of at least one qualifying record identified by the **select** subquery. Because the *exists-condition* uses the parenthesized **select** statement only to retrieve a record for comparison purposes, it requires only wildcard (*) field selection.

A predicate containing an *exists-condition* is true if the set of records specified by *select-expression* includes at least one record. If you add **not**, the predicate is true if there are *no* records that satisfy the subquery.

Format: exists-condition of Predicate

[**not**] **exists** (**select** * *where-clause*)

EXAMPLES

The following query displays all fields from CITIES records for which the POPULATION field is not missing:

```
QLI> select * from cities where population is not null;
```

The following query displays the CITY and STATE fields from cities with populations between 100,000 and 125,000:

```
QLI> select city, state from cities where population -
CON>   between 100000 and 125000
```

The following query displays all fields from STATES record in which the CAPITOL field contains the string ‘‘*ville*’’ preceded or followed by any number of characters:

```
QLI> select * from states where capitol like '%ville%';
```

SEE ALSO

select-expression (qli), *scalar-expression* (qli), **delete** (qli), **update** (qli)

NAME

prepare −prepare to commit transaction

SYNTAX

prepare [*database-handle-commalist*]

DESCRIPTION

The **prepare** command signals your intention to commit the default transaction.

The **prepare** command is particularly useful for sessions that access multiple databases. It executes the first phase of a two-phase commit. The access method polls all participants and waits for replies from each. It checks to see that no other database activity can affect the transaction. If the statement completes successfully, the database software guarantees that a **commit** statement will execute successfully if the disk is still intact.

If you are concerned with the internal operations associated with this statement, see the

ARGUMENTS

database-handle Specifies a name that can be used to qualify database reference when you are using multiple databases. A **prepare** command without the optional *database-handle* references all open databases. If you assign a database handle when you ready the database, you can use the handle to limit the scope of the **prepare** to specific databases. When you access more than one database in **qli**, the database software automatically starts up separate subtransactions for each database. However, these appear to be a single transaction. The optional *database-handle* lets you control these subtransactions explicitly by letting you prepare them by database.

If you forgot to assign a database handle when you readied the database and run into a problem with a database while you have several open, do not despair; **qli** automatically assigns a default handle if you do not. Type the following to find out the default database handle assigned by **qli**:

```
QLI> show databases
        Database "atlas.gdb" readied as QLI_0
        QLI>
```

Qli displays the names of all available entities, including databases and handles. The default handles are of the form ‘‘QLI_n,’’ where *n* is a numeric integer. Supply this handle as an argument to the **prepare** command:

```
QLI> prepare qli_1
        QLI>
```

EXAMPLE

The following statements ready several databases, perform some unspecified data manipulation, prepare to commit the transaction, and then commit the transaction:

```
QLI> ready remote_database_1.gdb
QLI> ready local_database.gdb
QLI> ready remote_database_2.gdb
   .
   .
   .
QLI> prepare
QLI> commit
QLI>
```

SEE ALSO

commit (qli)

DIAGNOSTICS

You may encounter the following message when you use the **prepare** statement:

- *expected database handle, encountered <string>*. You need a database handle. You may have mistyped the handle. Type *show databases* to check the database handle.

See also the discussion of errors in Chapter 1.

NAME

print −displaying records

SYNTAX

Standalone format:

print [*format-option-commalist*] [**distinct**]
{ *rse* | *value-expression-commalist* [**using** *edit-string*]
[(*query-header*)] **of** *rse* }
[**on** *'filespec'* | **to** *shell-command*]

format-option :== { **col** *integer* | **skip** *integer* }
query-header :== { *quoted-string-expression* | **-** }

For loop format:

for *rse*
print *value-expression-commalist*

DESCRIPTION

The **print** statement displays fields from records in a record stream. You can create the record stream in the **print** statement itself or in an outer **for** statement.

ARGUMENTS

distinct Prints only unique values (or combinations) of *value-expression*. For information about the project relational operation, see the description of the **reduce-clause** in the manual pages for *rse*.

value-expression **of** *rse* Specifies a list of fields or other values from the record stream created by the record selection expression. The value expressions can be formatted using an *edit string*. An edit string specifies an alphabetic, numeric, or date format for a field or computed value.

Syntax: Alphabetic and Miscellaneous Edit Strings

A	Any alphabetic character.
X	Any alphabetic character.
B	A blank space.
"quoted string" ’quoted string’	A string to be printed in the display.

Syntax: Numeric Edit Strings

9	An ordinary digit.
*	A leading asterisk (for checks).
Z	A leading digit, possibly blank-filled.
H	Hexadecimal representation of character.
+	Leading plus sign.
-	Leading minus sign.
$	Leading dollar sign.
(( ))	Parentheses around negative numbers.
DB	Debit.
CR	Credit.
.	Decimal point.
B	Blank space.
,	Comma for thousands, millions, etc.

Syntax: Date Edit Strings

Y (*integer*)	The year, from right to left. For 1986, *y(1)* yields *6*, *y(2)* yields *86*, and so on.
M (*integer*)	The month. The integer specifies how many of the characters in the month name to print.
N (*integer*)	The numeric month. The best value for the integer is 2.
D (*integer*)	The day of the month. The best value for the integer is 2.
W (*integer*)	The day of the week. The integer specifies how many of the characters in the day name to print.
B	A blank space.

query-header Specifies what **qli** prints at the head of a column. By default, it prints the field name, stacking the constituent words of the field name to fit the space. You can specify a quoted string to be printed, or a hyphen that suppresses the printing of a header. If the quoted string is too long for the field, you can cause **qli** to stack it up specifying quoted strings separated by slashes. For example, *"query"/"header"* will result in the word *query* appearing directly above the word *header* at the top of the column.

col *integer* Specifies in which column you want **qli** to start printing that field. The columns in this case refer to the screen positions on a terminal (that is, columns 1 to 80 or 132).

skip *integer* Specifies how many lines you want **qli** to skip between rows.

on *'filespec'* Sends the output to the named, quoted file, rather than writing it to your monitor.

to *shell-command* Sends the output to standard input of the shell command, rather than writing it to your screen. These commands typically send the output to a printer, as in **print**, **lpr**, **lpt**, **prf**, and **'prf -npag'**. Note that if you include a switch on the shell command, you must quote the entire command.

EXAMPLE

The following query prints all records in the STATES relation:

```
QLI> /* standalone, simple format */
QLI> print states sorted by state_name
QLI> The following query prints a literal value expression:
```

```
QLI> print "This is an utter outrage."
        This is an utter outrage.
```
The following query prints whatever you ask it to print:

```
QLI> print *."whatever your heart desires"
        Enter whatever your heart desires: chocolate
        chocolate
```
The following query writes field values from the STATES and SKI_AREAS to the file *shush_boom.dat*:

```
QLI> print state_name, name, city of states cross
        CON>  ski_areas over state on 'shush_boom.dat'
```
The following query prints field values from records in a stream created by a **for** command:

```
QLI> for states cross ski_areas over state
        CON>  print state_name, name, city
```
The following query prints the hexadecimal representation of Albany's altitude:

```
QLI> print altitude using hhhh of cities with city = 'Albany'
        .sp 0.25
         ALTITUDE
        ===========
             3b
```
The following query prints today's date using an edit string:

```
QLI> print "today" using w(8)" the "dd"th of "m(12)" in the year "y(4)
        Thursday the 15th of May in the year 1986
```
The following query prints the POPULATION field from CITIES using an edit string to format the number:

```
QLI> print city, population using z,zzz,zz9 of cities
```
The following commands print the numbers *-1* and *1* using an edit string:

```
QLI>
```

SEE ALSO

for (qli)

DIAGNOSTICS

You may encounter the following message when you use the **print** statement:

- *no items in print list*. You must provide a record selection expression or value expression.
- *Can't open output file*. **Qli** cannot open an output file for a *print on filespec* command.

See also the discussion of errors in Chapter 1.

NAME

quit −exiting qli

SYNTAX

quit

DESCRIPTION

The **quit** command ends a **qli** session and commits the current transaction.

Exit, **quit**, and the end-of-file character are exactly equivalent. The end-of-file characters are system-dependent:

- *Control-Z* for VAX/VMS, MicroVMS, and APOLLO DOMAIN
- *Control-D* for ULTRIX and SUN

EXAMPLE

```
QLI> quit
```

☞

SEE ALSO

exit (qli), **commit** (qli), **finish** (qli)

NAME

ready −access database

SYNTAX

ready *filespec* [**as** *database-handle*]

DESCRIPTION

The **ready** command attaches a database and opens it for access. This command must precede other database access in **qli**.

The **ready** command automatically starts a transaction that is not terminated until you commit it or roll it back. **Qli** automatically starts a new transaction with the next data manipulation statement that follows the **commit** or **rollback** command.

The database you access may be on another computer in the network. Such a database is called a *remote database*, and the computer where it is stored is called the *remote node*. The node you are using is called the *local node*. If you database you access is on the same node as you are, then it is a *local database*. To access a remote database, use the full network pathname of the database file or establish a logical link to it. Once you have readied the database, regardless of its location, you can read and write records to your heart's content.

ARGUMENTS

filespec Specifies the name of the file that contains the database. The file specification can contain the full pathname, including the name of the node on which the database is stored.

If the shell from which you invoked **qli** is case-sensitive, make sure that you type the name of the database file exactly as it appears when you list the directory.

If you are in a directory other than the one that contains the database file, *filespec* must include the pathname. If the database is on another node, the *filespec* must include the node name and pathname. You can define a link or logical name for the database file.

File specifications for remote databases have the following form:

Syntax: Remote Database File Specification

VMS to ULTRIX:
node-name::*filespec*

VMS to non-VMS and non-ULTRIX:
node-name^filespec

Within Apollo DOMAIN:
//*node-name*/*filespec*

All Else:
node-name:*filespec*

For example, the following command readies a database in the directory [*public.data*] on node *pariah*:

```
QLI> ready pariah:[public.data]phones.gdb
```

Make sure that what follows the colon is a valid file specification on the target system; use brackets, slashes, and spaces as appropriate.

database-handle Specifies a name that can be used to qualify database reference when you are using multiple databases. If you do not provide a handle, **qli** automatically assigns one of the form *qli_n*, where *n* represents a positive integer.

The optional *database-handle* lets you work with multiple databases, accessing each when you need it and closing each with a **finish** statement as appropriate. This approach saves system resources.

EXAMPLE

The following example readies a database for access:

```
QLI> ready atlas.gdb
```

The following example readies two databases for access, stating the explicit path to the database file for one database, and providing a database handle for each:

```
QLI> ready /usr/igor/datafiles/atlas.gdb as atlas
     QLI> ready mailing_list.gdb as mailing
          .
          .
          .
     QLI> finish atlas
     QLI> The following example readies a local and a remote database:
```

```
QLI> ready pariah:[doncikov.datafiles]atlas.gdb
     QLI> ready mailing_list.gdb
     QLI>
```

SEE ALSO

finish (qli)

DIAGNOSTICS

may not be able to find the database file you think you want to ready. The database file might not exist anymore, might not have the name you specified, or might not be where you thought it was. In any of these cases, check the database file name and location.

may not be able to ready a remote database due to a communication problem with the remote node. If that is the problem, make sure the remote servers are running.

You may encounter the following message when you use the **ready** command:

- *operating system directive failed*
  -no active servers (library/MBX manager)
  -communication error with journal "journal_directory_name"

  This message means that journaling has been enabled for the database you tried to ready, but no one has started the journal. Use **journal** to start the journal.

You may encounter the following message on an APOLLO:

- *Database error: I/O error during "ms_$mapl" operation for file "dbfile" -name not found (OS/naming server).*

  The database you tried to ready does not exist where you thought it did, is unavailable for some reason, or does not exist at all. Check the pathname and try again.

See also the discussion of errors in Chapter 1.

NAME

rename procedure −renaming a procedure

SYNTAX

rename procedure *old-name* [**to**] *new-name*

DESCRIPTION

The **rename procedure** statement changes the name of an existing procedure.

ARGUMENTS

old-name Specifies the name of the procedure you want to change.

new-name Specifies the new name of the procedure. The procedure name can be up to 31 characters and can contain alphabetic characters (A—Z and a—z, all stored as uppercase), numeric characters (0—9), underscores (_), and dollar signs ($). The procedure name must start with an alphabetic character.

EXAMPLE

The following command renames a procedure:

```
QLI> rename procedure capitol_info to capitol_city
     QLI>
```

SEE ALSO

See Chapter 8 of this manual for a complete discussion of procedures. See also the manual pages for **define procedure**, **edit procedure**, and **delete procedure** in this manual.

DIAGNOSTICS

You may encounter the following message when you use the **rename procedure** statement:

- *procedure name <name> is in use*.

  Choose another name.

- *procedure name over 31 characters*.

  Choose a shorter name.

- *gds_$create_blob failed*.

  The database software could not create the field in which the procedure text is stored. Try again.

- You may get the following errors when you execute a procedure:

  — *procedure <name> is undefined*.

  The procedure does not exist as specified. Type *show procedures* for a list of procedures.

  — *Procedure <name> not found*.

The procedure does not exist as specified. Type *show procedures* for a list of procedures.

See also the discussion of errors in Chapter 1.

NAME

repeat −repeat a statement

SYNTAX

repeat *integer-expression qli-statement*

DESCRIPTION

The **repeat** command lets you execute a **qli** statement multiple times.

If you want to include a procedure in a **repeat** command, enclose it in a **begin-end** statement. Otherwise, only the first statement in the procedure will be repeated.

If, at any time during the repeated operations, you decide to stop, type the end-of-file character (system-dependent). **Qli** then stops whatever it is doing and displays the following message:

```
Error: execution terminated by signal
```

Qli does not complete the operation that was interrupted. For example, suppose you decide to store five new SKI_AREAS. After storing two records, you begin providing the values for the third. However, you make a mistake while typing the value of the second field for the third record. You type the end-of-file character. **Qli** stores the first two records, but does not store the third record.

ARGUMENTS

integer-expression Specifies the number of repetitions. If *integer-expression* is not an integer, **qli** truncates the fractional part. The token *integer-expression* does not have to be a literal; instead, it can be a prompting expression.

qli-statement Any of the **qli** statements listed in Chapter 1. You can intermix the GDML and SQL variants of **qli** in a **repeat** command. However, as always, you cannot include a GDML statement in a SQL statement or vice versa.

EXAMPLE

The following example specifies that the **store** command is to be repeated five times, thereby causing **qli** to prompt for field values for five records:

```
QLI> repeat 5 store ski_areas
        .
        .
        .
```

QLI> The following statement prompts for the number of repetitions:

```
QLI> repeat *.'number of items'
     CON>  store ski_areas The following statement repeats a procedure five times:


QLI> repeat 5 begin :procedure end
          .
          .
          .
     QLI>
```

SEE ALSO

See Chapter 1 for a list of statements.

DIAGNOSTICS

See the discussion of errors in Chapter 1.

NAME

report −writing reports

SYNTAX

report *rse* [**on** *'filespec'* | **to** *shell-command*]
[**set report_name =** *value-expression*]
[**set columns** = *n*]
[**set lines** = *n*]
[**at top of report** [**print**] *value-expression*]
[**at bottom of report** [**print**] *value-expression*]
[**at top of page** [**print**] *value-expression*]
[**at top of** *database-field* **print** *value-expression*]...
[**at bottom of** *database-field* **print** *value-expression*]...
end_report [**on** *'filespec'* | **to** *shell-command*]

DESCRIPTION

The **report** command invokes **qli**'s report writer.

ARGUMENTS

rse Creates the record stream to be reported.

on *'filespec'* Sends the output to the named, quoted file, rather than writing it to your monitor. This clause can appear immediately after the **report** statement or after the **end_report** statement.

to *shell-command* Sends the output to standard input of the shell or command interpreter command, rather than writing it to your screen. These commands typically send the output to a printer, as in **print**, **lpr**, **lpt**, **prf**, and **'prf -npag'**. Note that if you include a switch on the shell command, you must quote the entire command. This clause can appear immediately after the **report** statement or after the **end_report** statement.

set report_name = *value-expression* Names the report.

set columns = *n* Specifies the width in mono-spaced characters for the output device. Reports printed on standard U.S. (8-½ by 11 inch) or European (A4) paper should not exceed 75 columns.

set lines = *n* Specifies the length in lines of the report. Reports printed on standard U.S. (8-½ by 11 inch) or European (A4) paper should not exceed 60 lines in length.

at top of report print *value-expression* Specifies a title to be printed at the beginning of the report.

at top of page print *value-expression* Specifies a title to be printed at the top of every page.

at bottom of page print *value-expression* Specifies a title to be printed at the bottom of every page.

at bottom of report print *value-expression* Specifies a title to be printed at the end of the report.

at top of *database-field* **print** *value-expression* Provides a control break and the title to print for that break. The *rse* for the report must include the *database-field* as a sort field.

at bottom of *database-field* **print** *value-expression* Provides a summary of a control group and an expression to print for that break. Typically, the **at bottom** matches an **at top** and calculates a total or aggregate expression for the control group.

EXAMPLE

The following statements report on records in the CITIES relation with control breaks by state:

```
QLI> report cities with population not missing sorted by state
CON>   set columns = 75
CON>   set lines = 55
CON>   set report_name = 'C I T I E S  B Y  S T A T E'
CON>   at top of state print state
CON>   print city, population, altitude, latitude, longitude
CON> end_report
QLI>
```

SEE ALSO

See Chapter 9 for a discussion of writing reports in **qli**.

DIAGNOSTICS

See the discussion of errors in Chapter 1.

NAME

restructure −move data from relation to relation

SYNTAX

[*database-handle.*]*relation-name = rse*

DESCRIPTION

The *restructure* statement lets you copy data from one relation to another and/or from one database to another. **Qli** automatically matches up fields and copies values from the older relation to the newer one.

See Chapter 5 for a discussion of restructuring databases.

ARGUMENTS

database_handle.relation_name Specifies the relation to which you want to assign values. The optional database handle is useful if you are using multiple databases with overlapping relation names.

rse Creates a record stream that serves as the source of values for *relation-name*.

EXAMPLE

The following example assumes that you have defined a new relation, CITY_STATES, that includes only cities with populations greater than 500,000. The new relation includes fields from both the CITIES and STATES relations.

```
QLI> city_states = cities cross states over state with population > 500000
```

SEE ALSO

assignment (qli)

DIAGNOSTICS

See the discussion of errors in Chapter 1.

NAME

rollback −undo changes made during transaction

SYNTAX

rollback [*database-handle-commalist*]

DESCRIPTION

The **rollback** command ends a transaction and undoes all changes made to the database since the most recent transaction started.

A **rollback** command without the optional *database-handle* affects all open databases. It causes the database software to undo all changes to data and metadata. **rollback** also flushes all modified buffers and closes any record streams that are open.

If you assign a database handle when you ready the database, you can use the handle to limit the effect of the **rollback** to specific databases. When you access more than one database in **qli**, the database software automatically starts up separate subtransactions for each database. However, these appear to be a single transaction. The optional *database-handle* lets you control these subtransactions explicitly by letting you commit or roll back transactions by database.

If you forgot to assign a database handle when you readied the database and run into a problem with a database while you have several open, do not despair; **qli** assigns a default handle if you have not specified one. Type the following to find out the default database handle assigned by **qli**:

```
QLI> show databases
        Database "atlas.gdb" readied as QLI_0
        QLI>
```

Qli displays the names of all available entities, including databases and handles. The default handles are of the form ‘‘QLI_n,’’ where *n* is a numeric integer. Supply this handle as an argument to the **rollback** command:

```
QLI> rollback qli_0
        QLI>
```

EXAMPLE

The following example performs some unspecified data manipulation activities and then undoes the changes, thus not writing them to the database:

```
QLI> ready atlas.gdb
                        .
                        .
                        .
        QLI> rollback
```

SEE ALSO

commit (qli), **finish** (qli), **prepare** (qli)

DIAGNOSTICS

A **rollback** cannot permitted to fail.

NAME

rse −search condition and other activities

SYNTAX

[*first-clause*] *record-source* [*with-clause*] [*reduced-clause*] [*sorted-clause*]

record-source ::= { *relation-clause* | *cross-source* }

relation-clause ::= [*context-variable in*] *relation-name*

cross-source ::= *relation-clause* **cross** *record-source*

DESCRIPTION

The *rse* (record selection expression) clause specifies the search and delivery conditions for record retrieval.

ARGUMENTS

first-clause Limits the records in a stream to the number you specify with an integer. The format of the *first-clause* follows:

Syntax: first-clause of RSE

first *integer*

Qli truncates any fractional portion of the integer. Unless you sort the record stream when you use the *first-clause*, the first *n* records are returned in random order.

relation-clause Identifies the target relation. The format of the *relation-clause* follows:

Syntax: relation-clause of RSE

[*context-variable* **in**] [*database-handle.*]*relation-name*

The optional *context-variable* is used for name recognition, and is associated with a relation. A context variable can contain up to 31 alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character.

Qli is not sensitive to the case of the context variable. For example, it treats **C** and **c** as the same character.

The optional *database-handle* identifies the database for multiple database access.

cross-clause Performs a join operation. The format of the *cross-clause* follows:

Syntax: relation-clause of RSE

cross *relation-clause* [**over** *field-name-commalist*]

The *cross-clause* creates dynamic relationships by matching up records from two or more different relations in the same database. The relationship can be based on the equality of common fields (equijoin), inequalities (non-equijoin), or where no relationship exists (cross product). Unlike most other *rse* clauses, *cross-clause* can be repeated to include as many relations as are necessary.

The **over** clause is semantically equivalent to a *with-clause* that equates a field in one relation with a field in another. The *field-name* must be exactly the same in both relations. Otherwise, you must use the *with-clause*, even if both fields are based on the same field.

with-clause Specifies a search condition or combination of search conditions. The format of the *with-clause* follows:

Syntax: with-clause of RSE

with *boolean-expression*

When you pass the search conditions to the access method, it evaluates the condition for each record that might possibly qualify. Conceptually, performs a record-by-record search, comparing the value you supplied with the value in the database field you specified. If the two values are in the relationship indicated by the operator you specified (for example, equals), the search condition evaluates to ‘‘true’’ and that record becomes part of the record stream. The search condition can result in a value of ‘‘true,’’ ‘‘false,’’ or ‘‘missing’’ for each record.

reduced-clause Performs a project operation, retrieving only the unique values for a field. The format of the *reduced-clause* follows:

Syntax: reduced-clause of RSE

reduced [**to**] *dbfield-expression-commalist* *dbfield-expression* ::= [*context-variable.*]*field-name*

When you ask for a record stream projected on a field, the access method considers a list of fields and eliminates records that do not have a unique combination of values for the listed fields.

sorted-clause Orders the output, returning the record stream sorted by the values of one or more sort keys. The format of the *sorted-clause* follows:

Syntax: sorted-clause of RSE

sorted [**by**] *sort-key-commalist*

sort-key ::= [**ascending** | **descending**] *dbfield-expression*

dbfield-expression ::= [*context-variable.*]*field-name*

You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *sort-clause* lets you have as many sort keys as you want.

Each sort key can specify whether the sorting order of the sort key is **ascending** (the default order for the first sort key) or **descending**. The sorting order is ‘‘sticky’’; that is, if you do not specify whether a particular sort key is **ascending** or **descending**, the database software assumes that you want the order specified for the most recent key. Therefore, if you list several sort keys, but only include the keyword **descending** for the first key, the database software sorts all keys in descending order.

EXAMPLES

The following query uses a *first-clause*, a *relation-clause*, and a *sorted-clause* to display the two ‘‘youngest’’ states:

```
QLI> for first 2 states sorted by descending statehood
CON>   print state_name | ' was admitted to the Union on ' | statehood
```

The following query uses two *relation-clause*s, a *cross-clause*, and a *sorted-clause* to list a ski area, city, and state in which it is located:

```
QLI> for s in states cross ski in ski_areas over state
CON>   print ski.name, ski.city, s.state_name
```

The following query does the same thing as the preceding query, but uses an explicitly qualified join condition in place of the **cross** shortcut:

```
QLI> for s in states cross ski in ski_areas with
CON>   s.state = ski.state
CON>   print ski.name, ski.city, s.state_name
```

The following query uses a *reduced-clause* to list the states in which there are ski areas:

```
QLI> print ski_areas reduced to state
```

The following query uses a *with-clause* to limit the disply to only those cities in Texas for which the value of the POPULATION field is not missing:

```
QLI> for cities with state = 'TX' and population not missing
CON>   print city, population, altitude
```

The following query displays the names of cities that are larger than the capitols of their

states:

```
. gdml_171a.epas
        QLI> for s in states cross c in cities over state cross
        CON>   cs in cities with cs.state = c.state and
        CON>   cs.city = s.capitol and
        CON>   cs.population < c.population
        CON>   sorted by s.state, c.city
        CON>   print c.city, s.state_name, ' is larger than ', s.capitol
```

The following statement displays only the names of states in which the capitol is not the largest city:

```
QLI> for s in states cross c in cities over state cross
        CON>   cs in cities with cs.state = c.state and
        CON>   cs.city = s.capitol and
        CON>   cs.population < c.population
        CON>   sorted by s.state
        CON>   reduced to s.state, s.capitol
        CON>   print s.state_name, ' contains cities larger than ', s.capitol
```

SEE ALSO

boolean-expression (qli), *value-expression* (qli)

NAME

scalar-expression −calculating value

SYNTAX

scalar-expression ::= [+ | -] *scalar-value* [*arithmetic-operator scalar-expression*]

scalar-value ::= { *field-expression* | *constant-expression* | *(scalar-expression*) }

arithmetic-operator ::= { + | - | * | / | | }

DESCRIPTION

The *scalar-expression* is a symbol or string of symbols used in predicates to calculate a value. The database software uses the result of the expression when executing the statement in which the expression appears.

You can add (+), subtract (-), multiply (*), and divide (/) scalar expressions. Arithmetic operations are evaluated in the normal order. You can use parentheses to change the order of evaluation.

The concatenation operator (|) is a formatting convenience. For example, if your **select** command includes a list of value expressions separated by commas, **qli** displays the field values in columnar order, padding out things like varying string fields with blanks. However, you can use the concatenation operator and constants to print a more legible display.

ARGUMENTS

field-expression References a database field. The format of the *field-expression* follows:

Syntax: field-expression of Scalar Expression

[*relation-name.* | *view-name.* | *alias.*]*database-field*

The optional *relation-name*, *view-name*, or *alias*, each followed by a required period (.), specifies the relation, view, or alias (synonym for a relation or view) in which the field is located. The alias is assigned to a relation or a view in a *select-expression*.

constant-expression A string of ASCII digits interpreted as a number or as a string of ASCII characters. The format of the *constant-expression* follows:

Syntax: constant-expression Scalar Expression

{ *integer-string* | *decimal-string* | *float-string* | *ascii-string* }

Integer numeric strings are written as signed or unsigned decimal integers without decimal points. For example, the following are integers: *-14*, *0*, *9*, and *+47*.

Decimal numeric strings are written as signed or unsigned decimal integers with decimal points. For example, the following are decimal strings: *-14.3*, *0.021*, *9.0*, and *+47.9*.

Floating numeric strings are written in scientific notation (that is, *E-format*). A number in scientific notation consists of a decimal string mantissa, the letter *E*, and a signed integer exponent. For example, the following are floating numerics: *7.12E+7* and *7.12E-7*.

Character strings are written using ASCII printing characters enclosed in single (') or double (") quotation marks. ASCII printing characters are:

- Uppercase alphabetic: *A—Z*
- Lowercase alphabetic: *a—z*
- Numerals: *0—9*
- Blank space and tab
- Special characters: ! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { } < > ; : ’ " \ | / ? . ,

EXAMPLES

The following query displays all fields from the CITIES record that represents Boston:

```
QLI> select * from cities where city = 'Boston'
```

The following query displays selected fields from the same record:

```
QLI> select population, altitude, latitude, longitude -
CON> from cities where city = 'Boston'
```

The following query displays selected fields from CITIES with a population greater than 1,000,000:

```
QLI> select city, state, population from cities -
CON> where population > 1000000
```

The following query joins records from the CITIES and STATES relations:

```
QLI> select c.city, s.state_name from cities c, states s -
CON> where s.state = c.state
```

SEE ALSO

predicate (qli)

NAME

select −selecting records

SYNTAX

select-statement ::= *select-expression* [*ordering-clause*]

ordering-clause ::= **order by** *sort-key-commalist*

sort-key ::= { **asc** | **desc** } { *database-field* | *integer* }

DECRIPTION

The **select** statement finds the record(s) of the relations specified in the **from** clause that satisfy the given search condition.

ARGUMENTS

select-expression Specifies the selection criteria. See the manual page for *select-expression*.

ordering-clause Returns the record stream sorted by the values of one or more *database-field*s.

You can sort a record stream alphabetically, numerically, by date, and by any combination of these. The *ordering-clause* lets you have up to 40 sort keys.

Each sort key can specify whether the sorting order of the sort key is **asc** (the default order for the first sort key) or **desc**. The sorting order is ‘‘sticky’’ that is, if you do not specify whether a particular sort key is **asc** or **desc**, it is assumed that you want the order specified for the most recent key. Therefore, if you list several sort keys, but only include the keyword **desc** for the first key, the database software sorts all keys in descending order.

EXAMPLE

The following query returns cities in Massachusetts:

```
QLI> select city, state, population
CON> from cities where state = 'MA';
```

The following query includes an *ordering-clause* with two sort keys:

```
QLI> select city, state, population -
CON> from cities where state = 'MA' order by -
CON> city, state;
```

The following example joins the relations CITIES and STATES on the basis of the equality of values in STATE:

```
QLI> select c.city, c.population, s.state_name, -
CON> from cities c, states s where -
CON> c.state = s.state order by s.state
```

SEE ALSO

select-expression (qli)

DIAGNOSTICS

You may encounter the following message when you use the **select** statement:

- *no items in print list.* You must provide something to print.

See also the discussion of errors in Chapter 1.

NAME

select-expression −selecting records

SYNTAX

select-clause [*where-clause*]

DESCRIPTION

The *select-expression* specifies the search and delivery conditions for record retrieval.

ARGUMENTS

select-clause Lists the fields to be returned and the source relation or view. The format of the *select-clause* follows:

Syntax: select-clause of Select Expression

select [**distinct**] *scalar-expression-commalist*
from *from-item-commalist*

from-item ::= *relation-name* [*alias*]

The optional keyword **distinct** specifies that only unique values are to be returned. The database software considers the values in the *scalar-expression* list and returns only one set value for each group of records that meets the selection criteria, and that have duplicate values for the *scalar-expression*.

The optional *alias* is used for name recognition, and is associated with a relation. An alias can contain up to 31 characters alphanumeric characters, dollar signs ($), and underscores (_). However, it must start with an alphabetic character. **Qli** is not sensitive to the case of the alias. For example, it treats **C** and **c** as the same character.

where-clause Specifies search conditions or combinations of search conditions. The format of the *where-clause* follows:

Syntax: where-clause of Select Expression

where *predicate*

When you specify a search condition or combination of conditions, the condition is evaluated for each record that might qualify . Conceptually, the database software performs a record-by-record search, comparing the value you supplied with the value in the database field you specified. If the two values satisfy the relationship you specified (for example, equals), the search condition evaluates to ‘‘true’’ and that record becomes part of the active set. The search condition can result in a value of ‘‘true,’’ ‘‘false,’’ or ‘‘missing’’ for each record. Such a statement, in which the choice is between the truth or falsity of a proposition, is called a ‘‘Boolean test’’ and is expressed by a *predicate*. See the manual page for *predicate*.

EXAMPLES

The following query projects the SKI_AREAS relation on the STATE field:

```
QLI> select distinct state from ski_areas;
```

The following query returns CITIES records for which the POPULATION field is not missing:

```
QLI> select city, state, population from cities -
CON> where population is not null
```

The following query joins two relations on the STATE field for cities whose population is not missing:

```
QLI> select c.city, s.state_name from cities c, states s -
CON> where c.state = s.state and c.population not missing
```

SEE ALSO

predicate (qli), *scalar-expression* (qli), **select** (qli)

NAME

set −toggling options

SYNTAX

set [**no**] { **blr** | **semicolon** | **statistics** }

DESCRIPTION

The **set** command lets you change various environmental features of **qli**.

ARGUMENTS

blr Displays the *binary language representation*, or *BLR*, of the query before displaying the results of the query.

You can use the **blr** switch to develop programs that use the call interface. For example, you can develop queries using **qli**, take the generated requests, and modify them as needed by your application. However, **qli** first parses the query for syntactic accuracy before sending off the request. If there is an error in your query, **qli** displays the appropriate error message and does not generate any BLR.

semicolon Changes **qli**'s line continuation behavior. Without the **semicolon** option set, you must break a command in the middle of a clause or at a comma, or use a hyphen. With **semicolon** set, **qli** does not execute a command until it encounters a semicolon. When you turn off this option, be sure that you type the semicolon at the end of the command:

```
QLI> set no semicolon;
```

statistics Displays system statistics after executing a query. You receive statistics on the following:

Number of read requests
Number of write requests
Number of requests for data which may be serviced in cache
Number of requests for updates which may be serviced in cache
Elapsed time
CPU time
Memory usage
Database page size
Database buffers used

EXAMPLE

The following commands set the **blr** switch and execute a query:

```
QLI> set blr
    QLI> print states
```

```
< display of STATES records >
0000 blr_version4,
< BLR for query is printed >
```

QLI> The following commands set the **statistics** switch and execute a query:

```
QLI> set statistics
```

```
QLI> print city, state of first 2 cities
< display of first 2 CITIES records >
Statistics for database "atlas.gdb"
    reads = 2 writes = 0 fetches = 7 marks = 0
    elapsed = 0.06 cpu = 0.05 mem = 55296
QLI>
```

SEE ALSO

show (qli)

DIAGNOSTICS

You may encounter the following message when you use the **set** command:

- *expected set option, encountered "invalid-option".* **Qli** did not recognize the **set** option you chose. Check the *Syntax* section above for the supported option.

See also the discussion of errors in Chapter 1.

NAME

shell −executing shell commands

SYNTAX

shell [’*shell-command*’]

DESCRIPTION

The **shell** command lets you execute shell commands from the **qli** environment. This command is supported only for ULTRIX, SUN, and DOMAIN environments. Use the **spawn** command on VMS systems.

ARGUMENTS

´shell-command´ A shell command enclosed in single (’) or double (") quotation marks.

If you do not issue a shell command, **qli** puts you in a shell. Type the end-of-file character to escape from the shell back to **qli**.

EXAMPLE

The following command escapes from **qli** and deposits you in a shell:

```
QLI> shell ’sh’
        ☞ Type the end-of-file character to return to qli.
```

The following command checks the time from within **qli**:

```
QLI> shell ’date’
        Tue Apr 23 13:54:22 EDT 1986
        QLI>
```

SEE ALSO

spawn (qli)

DIAGNOSTICS

You may encounter the following message when you use the **shell** command:

- *?(sh) "string" - name not found (OS/naming server).* The string you typed was not a command understood by the shell.

See also the discussion of errors in Chapter 1.

NAME

show −display information

SYNTAX

show { **all** | *database-handle* | **databases** | **fields** |
[**procedure**] *procedure-name* | **procedures** | **relations** |
relation-name | **system** [**relations**] | **variables** | **version** }

DESCRIPTION

The **show** command displays information.

If the **show** command references anything other than one of the listed options, **qli** assumes that you mean a procedure and looks for a procedure with that name. If it cannot find a procedure with that name, **qli** returns a message that the procedure was not found.

ARGUMENTS

all Displays the file specification and handle for all readied databases, relation name, and field names and datatype for each relation.

database-handle Displays everything about the named database. The handle must have been assigned in the **ready** statement, or may have been assigned automatically by **qli**.

databases Displays the file specification and handle of all readied databases.

fields [**for** *relation-name*] Displays all fields and datatypes for each relation in a readied database. If you specify a *relation-name*, it displays the fields and datatypes for the named relation.

procedure-name Displays the file specification of the database where *procedure-name* is stored and the text of the procedure.

This option is essentially the default. If you ask **qli** to show you something it does not understand, **qli** assumes that the desired item is a procedure.

procedures Displays the names of procedures for all readied databases.

relation-name Displays the field names and datatypes for the specified relation.

relations Displays the names of relations for each readied database.

system [**relations**] Displays the names of the system relations for each readied database.

variables Displays the names of declared variables.

version Displays release information about **qli** and the access method being used.

EXAMPLE

The following commands ready a database and then ask for information about all entities in the database:

```
QLI> ready atlas.gdb
 QLI> show all
 < display of all metadata information for readied database >
```

The following command asks for version information:

```
QLI> show version
      QLI, version "APL1.0F"
      Version(s) for database "atlas.gdb"
      < version information follows >
 QLI>
```

DIAGNOSTICS

You may encounter the following messages when you use the **show** command:

- *No databases are currently ready.*

  Qli cannot display anything because there is nothing to display. Ready a database and try the command again.

- *Procedure <procedure-name> not found.*

  Qli could not find a procedure with the name you typed. Type *show procedures* for a list of procedures. Likewise, if you reference a relation from a database other than the one(s) you have readied, **qli** assumes that the relation name is a procedure name. If it cannot find a procedure with that name, **qli** returns a message that the procedure was not found.

See also the discussion of errors in Chapter 1.

NAME

spawn −creating subprocess

SYNTAX

spawn

DESCRIPTION

The **spawn** command lets you ‘‘escape’’ from **qli** to a VMS subprocess. When you are finished with DCL commands, log out of the subprocess to return to **qli**.

EXAMPLE

The following command escapes from **qli** and deposits you at DCL level:

```
QLI> spawn
    $ Logout to return to qli.
```

DIAGNOSTICS

See the discussion of errors in Chapter 1.

NAME

store −insert new record

SYNTAX

store *relation-name* [**using** *assignment-statement*]

DESCRIPTION

The **store** statement inserts a new record into a relation.

Chapter 5 discusses assignment statements in detail.

ARGUMENTS

relation-name Specifies the relation into which you want to store a new record. If you specify only *relation-name* without an assignment, **qli** prompts you for field values.

assignment-statement A **qli** *assignment* statement.

If you specify *relation-name* and **using**, you can make assignments to fields using *assignment-statement*. In this case, you need the **begin-end** command if there is more than one field to which you must assign a value.

EXAMPLE

The following example stores a record using **qli**'s automatic prompting:

```
QLI> store ski_areas
        Enter NAME: Reedy Run
        Enter TYPE: N
        Enter CITY: Groton
        Enter STATE: MA
```

QLI> The following example stores a record, but uses a **begin-end** statement to structure a compound statement for assigning values to each field:

```
QLI> store ski_areas
        CON>  begin
        CON>     name = 'Moose Pond'
        CON>     type = 'N'
        CON>     city = 'Dixville Notch
        CON>     state = 'NH'
        CON>     end
        QLI>
```

SEE ALSO

See Chapter 5 for a discussion of assignments. See also the manual pages for *assignment* and **begin-end** in this manual.

DIAGNOSTICS

See the manual page for the *assignment* statement. See also the discussion of errors in Chapter 1.

NAME

then −sequencing statements

SYNTAX

qli-statement **then** *qli-statement*

DESCRIPTION

The **then** statement lets you sequence **qli** statements.

ARGUMENTS

qli-statement Any of the statements listed in Chapter 1 or a procedure.

EXAMPLE

The following example modifies a field value in several records, but displays each record before prompting for a new field value:

```
QLI> for ski_areas with state = 'VT'
CON> print then modify type
```

DIAGNOSTICS

You may encounter the following message when you use the **then** statement:

- *expected statement, encountered "command"*. You cannot use a command with the **then** statement.

See also the discussion of errors in Chapter 1.

NAME

update −modify field value

SYNTAX

update *relation-name*
set *assignment-commalist*
[**where** *predicate*]

assignment ::= *database-field = scalar-expression*

DESCRIPTION

The **update** statement changes the values of one or more fields in a record in a relation.

ARGUMENTS

relation-name Specifies the relation that contains the record you want to update.

assignment Assigns the *scalar-expression* to *database-field*.

predicate Selects the record to modify. If you provide a search condition with the optional *where-clause* of the *predicate*, updates the listed fields in the record(s) selected from *relation-name*. If you do not provide a search condition, updates all records in *relation-name*.

EXAMPLES

The following statement modifies the altitude of all cities:

```
QLI> update cities set altitude = altitude - 10
```

The following statement modifies the altitude of all cities in California and Washington:

```
QLI> update cities -
CON>   set altitude = altitude - 100 -
CON>   where state in ("CA", "WA");
```

The following statement elevates all cities in New York and changes the state code to ‘‘*NA*’’ (New Amsterdam):

```
QLI> update cities -
CON>   set altitude = altitude * 1.1, state = 'NA' -
CON>   where state = 'NY'
```

SEE ALSO

predicate (qli), **select** (qli)

DIAGNOSTICS

See the manual page for the *assignment* statement. See also the discussion of errors in Chapter 1.

NAME

value-expression −calculating value

SYNTAX

value-expression ::= { *arithmetic-expression* | *dbfield-expression* |
first-expression | *numeric-literal-expression* |
quoted-string-expression | *running-expression* |
statistical-expression | *(value-expression*) |
- *value-expression* }

DESCRIPTION

The *value-expression* is a symbol or string of symbols from which the database software calculates a value. The database software uses the result of the expression when executing the statement in which the expression appears.

ARGUMENTS

arithmetic-expression Combines value expressions and arithmetic operators. The format of the *arithmetic-expression* follows:

Syntax: arithmetic-expression Value Expression

value-expression-1 { + | - | * | / | | } *value-expression-2*

You can add (+), subtract (-), multiply (*), and divide (/) value expressions in assignment statements. Arithmetic operators are evaluated in the normal order. Use parentheses to change the order of evaluation.

You can use the concatenation operator (|) to combine field values in record selection expressions.

dbfield-expression References database fields. This expression can occur in several clauses of *rse* and *boolean-expression*. The format of the *dbfield-expression* follows:

Syntax: dbfield-expression Value Expression

[*context-variable.*]*field-name*

The optional *context-variable* lets you qualify the database field for multi-relation operations. You must declare a context variable for a relation in the *relation-clause* of the record selection expression.

first-expression Forms a record stream and evaluates an expression. The format of the *first-expression* follows:

Syntax: first-expression Value Expression

first *value-expression-1* **from** *rse*
[**else** *value-expression-2*]

The database software finds the first qualifying record in the record stream. If the stream is empty, it returns an error unless you supply an **else** clause. Otherwise, the database software evaluates *value-expression-2* in the context of the record it found. The result of the evaluation is returned as the value of *first-expression* or the value specified in the **else** clause.

If you use the *first-expression* in a **print** command, you must enclose it in parentheses. Otherwise, **qli** assumes that you are using the *first-clause* of the record selection expression.

numeric-literal-expression Represents a decimal number as a string of digits with an optional decimal point. The format of the *numeric-literal-expression* follows:

Syntax: numeric-literal-expression Value Expression

[+ | -] *string*[.*string*]

quoted-string-expression A string of ASCII characters enclosed in single (') or double (") quotation marks. The format of the *quoted-string-expression* follows:

Syntax: quoted-string Value Expression

"string"

ASCII printing characters are:

- Uppercase alphabetic: *A—Z*
- Lowercase alphabetic: *a—z*
- Numerals: *0—9*
- Special characters: ! @ # $ % ^ & * ( ) _ - + = ‘ ~ [ ] { } < > ; : ’ " \ | / ? . ,

running-expression Calculates a running count for a record stream or a running total for an expression. The format of the *running-expression* follows:

Syntax: running Value Expression

{ **running count** | **running total** *value-expression* }

statistical-expression Calculates a value based on a value expression. The format of the *statistical-expression* follows:

Syntax: statistical-expression Value Expression

{ *statistical-operation value-expression* **of** *rse* |
count **of** *rse* }

statistical-operation ::= { **average** | **max** | **min** | **total** }

If a field value included in *value-expression* is missing for a record, that record is not included in the calculation. For **average**, **max**, and **min**, if the record stream created by *rse* is empty, the value of the statistical expression is missing. For **total** and **count**, if the record stream is empty, the total is 0.

EXAMPLES

The following statement uses *dbfield-expression*s to display the city and state, an *arithmetic-expression* that calculates and displays the altitude in meters, a *numeric-literal-expression* (0.3048) used in the arithmetic operation, and two *quoted-string-expression*s:

```
QLI> for c in cities cross s in states over state
CON>   print c.city, s.state_name | ' is situated at ' |
CON>   c.altitude * 0.3048 | ' meters above sea level.'
```

SEE ALSO

boolean-expression (qli), *rse* (qli)