Chapter 7. SQL Procedures

An SQL procedure consists of a CREATE PROCEDURE statement with a procedure body.

This chapter contains the syntax and parameter descriptions for the procedure body in an SQL Procedure Statement.

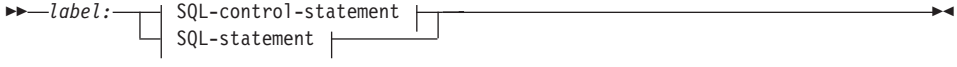
SQL Procedure Statement

The procedure body in an SQL stored procedure definition contains the source statements for the stored procedure.

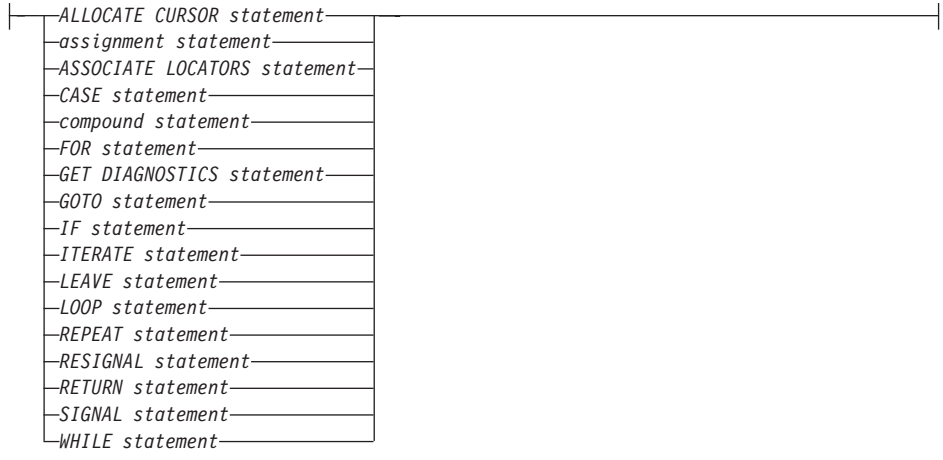
This chapter contains syntax diagrams, semantic descriptions, rules, and examples of the use of the statements that constitute the procedure body.

If an SQL control statement is specified as the SQL procedure body, multiple statements can be specified within the control statement. These statements are defined as SQL procedure statements.

Syntax



SQL-control-statement:



Description

label:
Specifies the label for an SQL procedure statement. The label must be unique within a list of SQL procedure statements, including any compound statements nested within the list. Note that compound statements that are not nested may use the same label. A list of SQL procedure statements is possible in a number of SQL control statements.

SQL-statement
All executable SQL statements can be contained within an SQL procedure body with the exception of the following:

- CONNECT
- CREATE any object other than indexes, tables, or views
- DESCRIBE
- DISCONNECT
- DROP any object other than indexes, tables, or views
- FLUSH EVENT MONITOR
- REFRESH TABLE
- RELEASE (connection only)
- RENAME TABLE
- RENAME TABLESPACE
- REVOKE
- SET CONNECTION
- SET INTEGRITY

Note: You may include CALL statements within an SQL procedure body, but these CALL statements can only call another SQL procedure. CALL statements within an SQL procedure body cannot call other types of stored procedures.

ALLOCATE CURSOR Statement

The ALLOCATE CURSOR statement allocates a cursor for the result set identified by the result set locator variable. See "ASSOCIATE LOCATORS Statement" on page 1066 for more information on result set locator variables.

Syntax

Description

cursor-name
Specifies the cursor name. The name must not identify a cursor that has already been declared in the source SQL procedure (SQLSTATE 24502).

CURSOR FOR RESULT SET *rs-locator-variable*

Specifies a result set locator variable that has been declared in the source SQL procedure, according to the rules for host variables. For more information on declaring SQL variables, see "SQL variable declaration" on page 1071.

The result set locator variable must contain a valid result set locator value, as returned by the ASSOCIATE LOCATORS SQL statement (SQLSTATE 24501).

Notes

- **Dynamically prepared ALLOCATE CURSOR statements**: The EXECUTE statement with the USING clause must be used to execute a dynamically prepared ALLOCATE CURSOR statement. In a dynamically prepared statement, references to host variables are represented by parameter markers (question marks).

  In the ALLOCATE CURSOR statement, *rs-locator-variable* is always a host variable. Thus, for a dynamically prepared ALLOCATE CURSOR statement, the USING clause of the EXECUTE statement must identify the host variable whose value is to be substituted for the parameter marker that represents *rs-locator-variable*.
- You cannot prepare an ALLOCATE CURSOR statement with a statement identifier that has already been used in a DECLARE CURSOR statement. For example, the following SQL statements are invalid because the PREPARE statement uses `STMT1` as an identifier for the ALLOCATE CURSOR statement and `STMT1` has already been used for a DECLARE CURSOR statement.

```
DECLARE CURSOR C1 FOR STMT1;

PREPARE STMT1 FROM
   'ALLOCATE C2 CURSOR FOR RESULT SET ?';
```

Rules

- The following rules apply when using an allocated cursor:
  - An allocated cursor cannot be opened with the OPEN statement (SQLSTATE 24502).
  - An allocated cursor can be closed with the CLOSE statement. Closing an allocated cursor closes the associated cursor in the stored procedure.
  - Only one cursor can be allocated to each result set.
- Allocated cursors last until a rollback operation, an implicit close, or an explicit close.
- A commit operation destroys allocated cursors that are not defined WITH HOLD by the stored procedure.
- Destroying an allocated cursor closes the associated cursor in the SQL procedure.

Examples

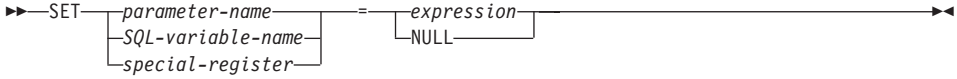
This SQL procedure example defines and associates cursor C1 with the result set locator variable LOC1 and the related result set returned by the SQL procedure:

```
ALLOCATE C1 CURSOR FOR RESULT SET LOC1;
```

Assignment Statement

The assignment statement assigns a value to an output parameter, a local variable, or a special register.

Syntax



Description

parameter-name
Identifies the parameter that is the assignment target. The parameter must be specified in *parameter-declaration* in the CREATE PROCEDURE statement and must be defined as an OUT or INOUT parameter.

SQL-variable-name
Identifies the SQL variable that is the assignment target. SQL variables must be declared before they are used. SQL variables can be defined in a compound statement.

special-register
Identifies the special register that is the assignment target. If the special register accepts a schema name as a value, including the CURRENT FUNCTION PATH or CURRENT SCHEMA special registers, DB2 determines whether the assignment parameter is an SQL variable. If the assignment parameter is an SQL variable, DB2 assigns the value of the SQL variable to the special register. If the assignment parameter is not an SQL variable, DB2 assumes that the assignment parameter is a schema name and assigns that schema name to the special register.

The initial settings of special register values in an SQL procedure are inherited from the caller of the procedure. The assignment of a new setting is valid for the entire SQL procedure where it is set, and will be inherited by any procedure that it subsequently calls. When a procedure returns to its caller, the special registers are restored to the original settings of the caller.

expression **or NULL**
Specifies the expression or value that is the source for the assignment.

Rules

- Assignment statements in SQL procedures must conform to the SQL assignment rules.
- The data type of the target and source must be compatible.

- When a string is assigned to a fixed-length variable and the length of the string is less than the length attribute of the target, the string is padded on the right with the necessary number of single-byte, double-byte, or UCS-2 blanks.
- When a string is assigned to a variable and the string is longer than the length attribute of the variable, an error is issued.
- A string assigned to a variable is first converted, if necessary, to the codepage of the target.
- If truncation of the whole part of the number occurs on assignment to a numeric variable, an error is raised.

Examples

Increase the SQL variable `p_salary` by 10 percent.

```
SET p_salary = p_salary + (p_salary * .10)
```

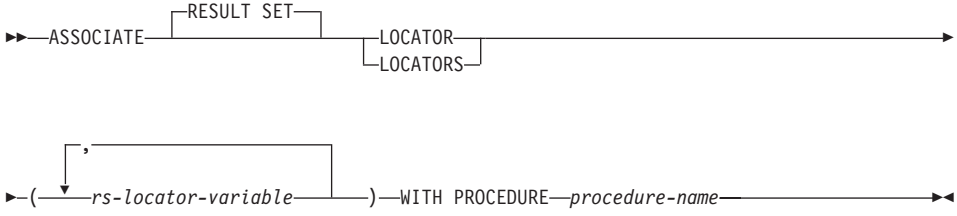
Set SQL variable `p_salary` to the null value.

```
SET p_salary = NULL
```

ASSOCIATE LOCATORS Statement

The ASSOCIATE LOCATORS statement gets the result set locator value for each result set returned by a stored procedure.

Syntax



Description

rs-locator-variable
Specifies a result set locator variable that has been declared in a compound statement.

WITH PROCEDURE
Identifies the stored procedure that returns result set locators by the specified procedure name.

procedure-name
A procedure name is a qualified or unqualified name. Each part of the name must be composed of SBCS characters.

A fully qualified procedure name is a two-part name. The first part is an identifier that contains the schema name of the stored procedure. The last part is an identifier that contains the name of the stored procedure. A period must separate each of the parts. Any or all of the parts can be a delimited identifier.

If the procedure name is unqualified, it has only one name because the implicit schema name is not added as a qualifier to the procedure name. Successful execution of the ASSOCIATE LOCATOR statement only requires that the unqualified procedure name in the statement is the same as the procedure name in the most recently executed CALL statement that was specified with an unqualified procedure name. The implicit schema name for the unqualified name in the CALL statement is not considered in the match. The rules for how the procedure name must be specified are described below.

When the ASSOCIATE LOCATORS statement is executed, the procedure name or specification must identify a stored procedure that the requester has already invoked using the CALL statement. The procedure name in the ASSOCIATE LOCATORS statement must be specified the same way

that it was specified on the CALL statement. For example, if a two-part name was specified on the CALL statement, you must use a two-part name in the ASSOCIATE LOCATORS statement.

Rules

- More than one locator can be assigned to a result set. You can issue the same ASSOCIATE LOCATORS statement more than once with different result set locator variables.
- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is less than the number of locators returned by the stored procedure, all variables in the statement are assigned a value, and a warning is issued.
- If the number of result set locator variables that are listed in the ASSOCIATE LOCATORS statement is greater than the number of locators returned by the stored procedure, the extra variables are assigned a value of 0.
- If a stored procedure is called more than once from the same caller, only the most recent result sets are accessible.

Examples

The statements in the following examples are assumed to be embedded in SQL Procedures.

Example 1: Use result set locator variables LOC1 and LOC2 to get the result set locator values for the two result sets returned by stored procedure P1. Assume that the stored procedure is called with a one-part name.

```
CALL P1;
ASSOCIATE RESULT SET LOCATORS (LOC1, LOC2)
   WITH PROCEDURE P1;
```

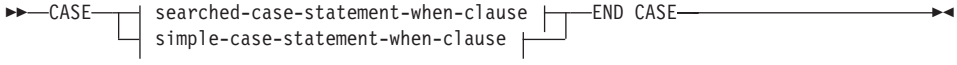
Example 2: Repeat the scenario in Example 1, but use a two-part name to specify an explicit schema name for the stored procedure to ensure that stored procedure P1 in schema MYSCHEMA is used.

```
CALL MYSCHEMA.P1;
ASSOCIATE RESULT SET LOCATORS (LOC1, LOC2)
   WITH PROCEDURE MYSCHEMA.P1;
```

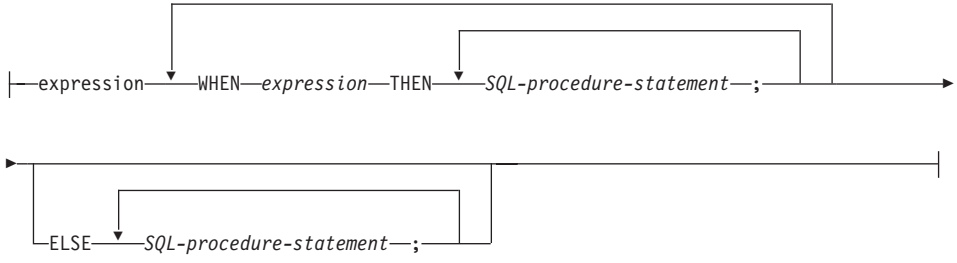
CASE Statement

The CASE statement selects an execution path based on multiple conditions.

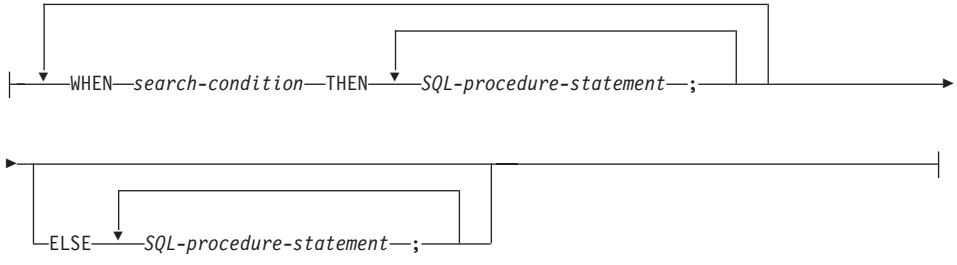
Syntax



simple-case-statement-when-clause:



searched-case-statement-when-clause:



Description

CASE
Begins a *case-expression*.

simple-case-statement-when-clause
The value of the *expression* prior to the first WHEN keyword is tested for equality with the value of each *expression* that follows the WHEN keyword. If the search condition is true, the THEN statement is executed. If the result is unknown or false, processing continues to the next search condition. If the result does not match any of the search conditions, and an ELSE clause is present, the statements in the ELSE clause are processed.

searched-case-statement-when-clause
The *search-condition* following the WHEN keyword is evaluated. If it evaluates to true, the statements in the associated THEN clause are processed. If it evaluates to false, or unknown, the next *search-condition* is evaluated. If no *search-condition* evaluates to true and an ELSE clause is present, the statements in the ELSE clause are processed.

SQL-procedure-statement
Specifies a statement that should be invoked.

END CASE
Ends a *case-statement*.

Notes

- If none of the conditions specified in the WHEN are true, and an ELSE clause is not specified, an error is issued at runtime, and the execution of the case statement is terminated (SQLSTATE 20000).
- Ensure that your CASE statement covers all possible execution conditions.

Examples

Depending on the value of SQL variable `v_workdept`, update column DEPTNAME in table DEPARTMENT with the appropriate name.

The following example shows how to do this using the syntax for a *simple-case-statement-when-clause*:

```
CASE v_workdept
   WHEN'A00'
     THEN UPDATE department
     SET deptname = 'DATA ACCESS 1';
   WHEN 'B01'
     THEN UPDATE department
     SET deptname = 'DATA ACCESS 2';
   ELSE UPDATE department
     SET deptname = 'DATA ACCESS 3';
END CASE
```

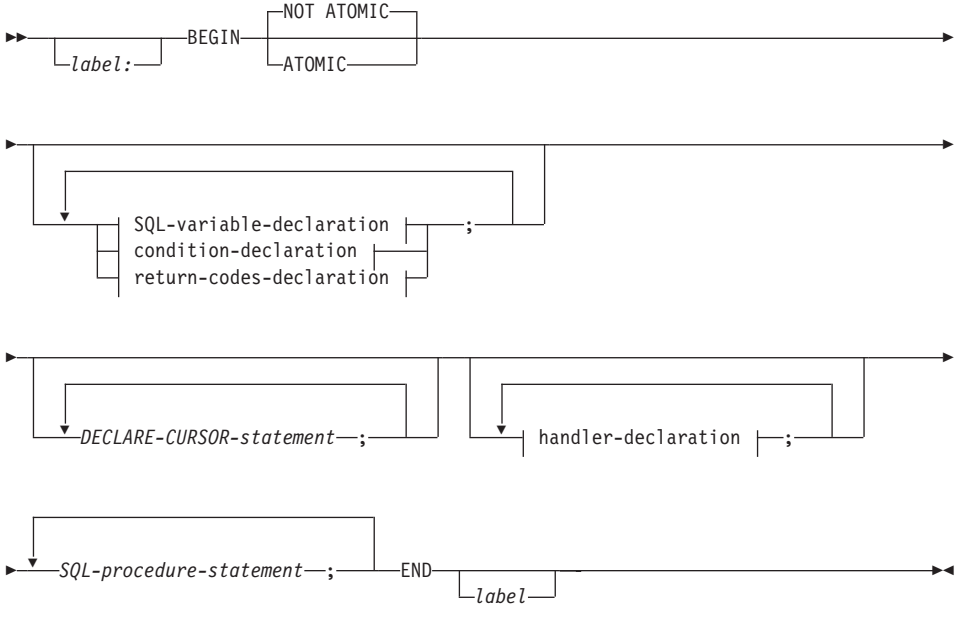
The following example shows how to do this using the syntax for a *searched-case-statement-when-clause*:

```
CASE
   WHEN v_workdept = 'A00'
     THEN UPDATE department
     SET deptname = 'DATA ACCESS 1';
   WHEN v_workdept = 'B01'
     THEN UPDATE department
     SET deptname = 'DATA ACCESS 2';
   ELSE UPDATE department
     SET deptname = 'DATA ACCESS 3';
END CASE
```

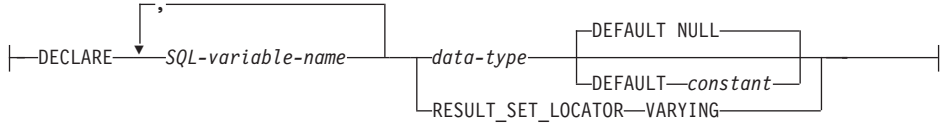
Compound Statement

A compound statement groups other statements together in an SQL procedure. You can declare SQL variables, cursors, and condition handlers within a compound statement.

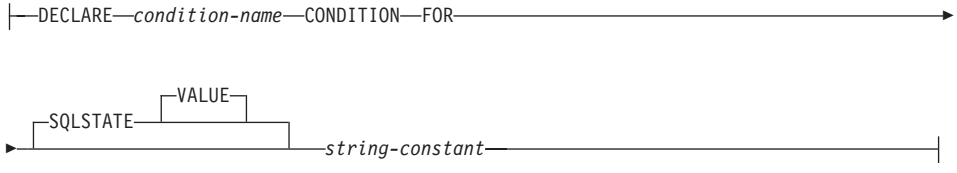
Syntax



SQL-variable-declaration:

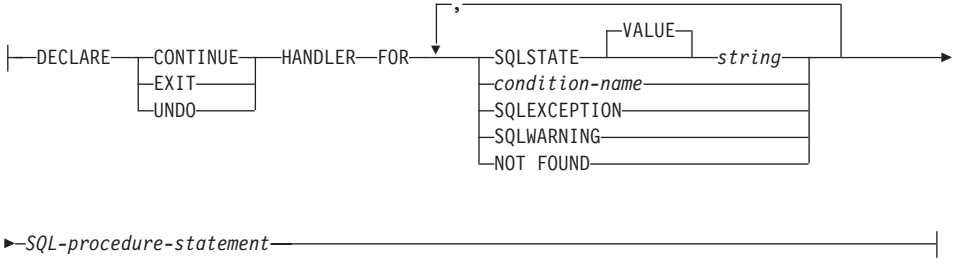


condition-declaration:



return-codes-declaration:

handler-declaration:



Description

label

Defines the label for the code block. If the beginning label is specified, it can be used to qualify SQL variables declared in the compound statement and can also be specified on a LEAVE statement. If the ending label is specified, it must be the same as the beginning label.

ATOMIC or NOT ATOMIC

ATOMIC indicates that if an error occurs in the compound statement, all SQL statements in the compound statement will be rolled back. NOT ATOMIC indicates that an error within the compound statement does not cause the compound statement to be rolled back.

SQL-variable-declaration

Declares a variable that is local to the compound statement.

SQL-variable-name

Defines the name of a local variable. DB2 converts all SQL variable names to uppercase. The name cannot be the same as another SQL variable within the same compound statement and cannot be the same as a parameter name. SQL variable names should not be the same as column names. If an SQL statement contains an identifier with the same name as an SQL variable and a column reference, DB2 interprets the identifier as a column.

data-type

Specifies the data type of the variable. Refer to “Data Types” on page 75 for a description of SQL data types. User-defined data types, graphic types, and FOR BIT DATA are not supported.

DEFAULT *constant* **or NULL**
Defines the default for the SQL variable. The variable is initialized when the SQL procedure is called. If a default value is not specified, the variable is initialized to NULL.

RESULT_SET_LOCATOR VARYING
Specifies the data type for a result set locator variable.

condition-declaration
Declares a condition name and corresponding SQLSTATE value.

condition-name
Specifies the name of the condition. The condition name must be unique within the procedure body and can be referenced only within the compound statement in which it is declared.

FOR SQLSTATE *string-constant*
Specifies the SQLSTATE that is associated with the condition. The string-constant must be specified as five characters enclosed in single quotes, and cannot be '00000'.

return-codes-declaration
Declares special variables called SQLSTATE and SQLCODE that are set automatically to the value returned after processing an SQL statement. Both the SQLSTATE and SQLCODE variables can only be declared in the outermost compound statement of the SQL procedure body. These variables may be declared only once per SQL procedure.

declare-cursor-statement
Declares a cursor in the procedure body. Each cursor must have a unique name. The cursor can be referenced only from within the compound statement. Use an OPEN statement to open the cursor, and a FETCH statement to read rows using the cursor. To return result sets from the SQL procedure to the client application, the cursor must be declared using the WITH RETURN clause. The following example returns one result set to the client application:

```
CREATE PROCEDURE RESULT_SET()
   LANGUAGE SQL
   RESULT SETS 1
   BEGIN
      DECLARE C1 CURSOR WITH RETURN FOR
         SELECT id, name, dept, job
            FROM staff;
      OPEN C1;
   END
```

Note: To process result sets, you must write your client application using one of the DB2 Call Level Interface (DB2 CLI), Open Database Connectivity (ODBC), Java Database Connectivity (JDBC), or embedded SQL for Java (SQLJ) application programming interfaces.

For more information on declaring a cursor, refer to "DECLARE CURSOR" on page 841.

handler-declaration
Specifies a *handler*, a set of statements to execute when an exception or completion condition occurs in the compound statement. *SQL-procedure-statement* is a statement that executes when the handler receives control.

A handler is active only within the compound statement in which it is declared.

There are three types of condition handlers:

CONTINUE
After the handler is invoked successfully, control is returned to the SQL statement that follows the statement that raised the exception. If the error that raised the exception is a FOR, IF, CASE, WHILE, or REPEAT statement (but not an SQL-procedure-statement within one of these), then control returns to the statement that follows END FOR, END IF, END CASE, END WHILE, or END REPEAT.

EXIT
After the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler.

UNDO
Before the handler is invoked, any SQL changes that were made in the compound statement are rolled back. After the handler is invoked successfully, control is returned to the end of the compound statement that declared the handler. If UNDO is specified, then ATOMIC must be specified.

The conditions under which the handler is activated:

SQLSTATE *string*
Specifies an SQLSTATE for which the handler is invoked. The SQLSTATE cannot be '00000'.

condition-name
Specifies a condition name for which the handler is invoked. The condition name must be previously defined in a condition declaration.

SQLEXCEPTION
Specifies that the handler is invoked when an SQLEXCEPTION occurs. An SQLEXCEPTION is an SQLSTATE where the first two characters are not "00", "01", or "02".

SQLWARNING
Specifies that the handler is invoked when an SQLWARNING occurs. An SQLWARNING is an SQLSTATE where the first two characters are "01".

NOT FOUND
Specifies that the handler is invoked when a NOT FOUND condition occurs. NOT FOUND corresponds to an SQLSTATE where the first two characters are "02".

Rules

- ATOMIC compound statements cannot be nested.
- The following rules apply to handler declarations:
  - A handler declaration that contains SQLEXCEPTION, SQLWARNING, or NOT FOUND cannot contain additional SQLSTATE or condition names.
  - Handler declarations within the same compound statement cannot contain duplicate conditions.
  - A handler declaration cannot contain the same condition code or SQLSTATE value more than once, and cannot contain an SQLSTATE value and a condition name that represent the same SQLSTATE value. For a list of SQLSTATE values and more information, refer to the *Message Reference*.
  - A handler is activated when it is the most appropriate handler for an exception or completion condition. The most appropriate handler is a handler (for the exception or completion condition) that is defined in the compound statement, nearest in scope to the statement with the exception or completion condition. If an exception occurs for which there is no handler, execution of the compound statement is terminated.

Examples

Create a procedure body with a compound statement that performs the following actions:

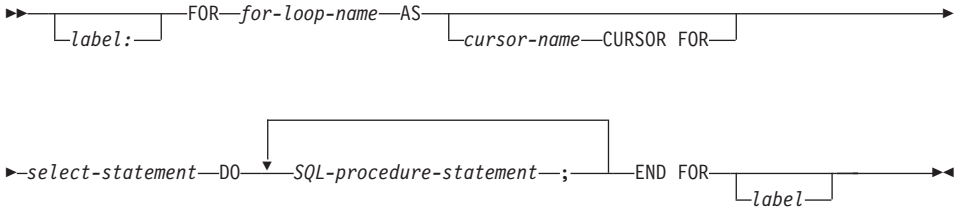
1. Declares SQL variables
2. Declares a cursor to return the salary of employees in a department determined by an IN parameter. In the SELECT statement, casts the data type of the *salary* column from a DECIMAL into a DOUBLE.
3. Declares an EXIT handler for the condition NOT FOUND (end of file) which assigns the value '6666' to the OUT parameter `medianSalary`
4. Select the number of employees in the given department into the SQL variable `numRecords`
5. Fetch rows from the cursor in a WHILE loop until 50% + 1 of the employees have been retrieved
6. Return the median salary

```
CREATE PROCEDURE DEPT_MEDIAN
   (IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
   LANGUAGE SQL
   BEGIN
    DECLARE v_numRecords INTEGER DEFAULT 1;
    DECLARE v_counter INTEGER DEFAULT 0;
    DECLARE c1 CURSOR FOR
       SELECT CAST(salary AS DOUBLE) FROM staff
          WHERE DEPT = deptNumber
          ORDER BY salary;
    DECLARE EXIT HANDLER FOR NOT FOUND
       SET medianSalary = 6666;
-- initialize OUT parameter
       SET medianSalary = 0;
       SELECT COUNT(*) INTO v_numRecords FROM staff
          WHERE DEPT = deptNumber;
       OPEN c1;
       WHILE v_counter < (v_numRecords / 2 + 1) DO
          FETCH c1 INTO medianSalary;
          SET v_counter = v_counter + 1;
       END WHILE;
       CLOSE c1;
    END
```

FOR Statement

The FOR statement executes a statement or group of statements for each row of a table.

Syntax



Description

label
Specifies the label for the FOR statement. If the beginning label is specified, that label can be used in LEAVE and ITERATE statements. If the ending label is specified, it must be the same as the beginning label.

for-loop-name
Specifies a label for the implicit compound statement generated to implement the FOR statement. It follows the rules for the label of a compound statement except that it cannot be used with and ITERATE or LEAVE statement within the FOR statement. The *for-loop-name* is used to qualify the column names returned by the specified *select-statement*.

cursor-name
Names the cursor that is used to select rows from the result table from the SELECT statement. If not specified, DB2 generates a unique cursor name.

select-statement
Specifies the SELECT statement of the cursor. All columns in the select list must have a name and there cannot be two columns with the same name.

SQL-procedure-statement
Specifies a statement (or statements) to be invoked for each row of the table.

Rules

- The select list must consist of unique column names and the table specified in the select list must exist when the procedure is created, or it must be a table created in a previous SQL procedure statement.
- The cursor specified in a for-statement cannot be referenced outside the for-statement and cannot be specified in an OPEN, FETCH, or CLOSE statement.

Examples

In the following example, the for-statement is used to iterate over the entire `employee` table. For each row in the table, the SQL variable `fullname` is set to the last name of the employee, followed by a comma, the first name, a blank space, and the middle initial. Each value for `fullname` is inserted into table `tnames`.

```
BEGIN
   DECLARE fullname CHAR(40);
   FOR v1 AS
      SELECT firstnme, midinit, lastname FROM employee
         DO
         SET fullname = lastname || ',' || firstnme ||' ' || midinit;
         INSERT INTO tnames VALUE (fullname);
   END FOR
END
```

GET DIAGNOSTICS Statement

The GET DIAGNOSTICS statement obtains information about the previous SQL statement invoked.

Syntax

Description

SQL-variable-name
Identifies the variable that is the assignment target. The variable must be an integer variable. SQL variables can be defined in a compound statement.

ROW_COUNT
Identifies the number of rows associated with the previous SQL statement that was invoked. If the previous SQL statement is a DELETE, INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows deleted, inserted, or updated by that statement, excluding rows affected by either triggers or referential integrity constraints. If the previous statement is a PREPARE statement, ROW_COUNT identifies the *estimated* number of result rows in the prepared statement.

RETURN_STATUS
Identifies the status value returned from the stored procedure associated with the previously executed SQL statement, provided that the statement was a CALL statement invoking a procedure that returns a status. If the previous statement is not such a statement, the value returned has no meaning and could be any integer.

Rules

- The GET DIAGNOSTICS statement does not change the contents of the diagnostics area (SQLCA). If an SQLSTATE or SQLCODE special variable is declared in the SQL procedure, these are set to the SQLSTATE or SQLCODE returned from issuing the GET DIAGNOSTICS statement.

Examples

In an SQL procedure, execute a GET DIAGNOSTICS statement to determine how many rows were updated.

```
CREATE PROCEDURE sqlprocg (IN deptnbr VARCHAR(3))
   LANGUAGE SQL
   BEGIN
      DECLARE SQLSTATE CHAR(5);
      DECLARE rcount INTEGER;
      UPDATE CORPDATA.PROJECT
         SET PRSTAFF = PRSTAFF + 1.5
```

```
            WHERE DEPTNO = deptnbr;
          GET DIAGNOSTICS rcount = ROW_COUNT;
-- At this point, rcount contains the number of rows that were updated.
    ...
       END
```

Within an SQL procedure, handle the returned status value from the invocation of a stored procedure called TRYIT that could either explicitly RETURN a positive value indicating a user failure, or encounter SQL errors that would result in a negative return status value. If the procedure is successful, it returns a value of zero.

```
CREATE PROCEDURE TESTIT ()
  LANGUAGE SQL
  A1:BEGIN
     DECLARE RETVAL INTEGER DEFAULT 0;
     ...
     CALL TRYIT;
     GET DIAGNOSTICS RETVAL = RETURN_STATUS;
     IF RETVAL <> 0 THEN
        ...
        LEAVE A1;
     ELSE
        ...
     END IF;
   END A1
```

GOTO Statement

The GOTO statement is used to branch to a user-defined label within an SQL routine.

Syntax

Description

label

Specifies a labelled statement where processing is to continue. The labelled statement and the GOTO statement must be in the same scope:

- If the GOTO statement is defined in a FOR statement, *label* must be defined inside the same FOR statement, excluding a nested FOR statement or nested compound statement
- If the GOTO statement is defined in a compound statement, *label* must be defined inside the same compound statement, excluding a nested FOR statement or nested compound statement
- If the GOTO statement is defined in a handler, *label* must be defined in the same handler, following the other scope rules
- If the GOTO statement is defined outside of a handler, *label* must not be defined within a handler.

If *label* is not defined within a scope that the GOTO statement can reach, an error is returned (SQLSTATE 42736).

Rules

- It is recommended that the GOTO statement be used sparingly. This statement interferes with normal processing sequences, thus making a routine more difficult to read and maintain. Before using a GOTO statement, determine whether another statement, such as IF or LEAVE, can be used in place, to eliminate the need for a GOTO statement.

Examples

In the following compound statement, the parameters *rating* and *v_empno* are passed into the procedure, which then returns the output parameter *return_parm* as a date duration. If the employee's time in service with the company is less than 6 months, the GOTO statement transfers control to the end of the procedure, and *new_salary* is left unchanged.

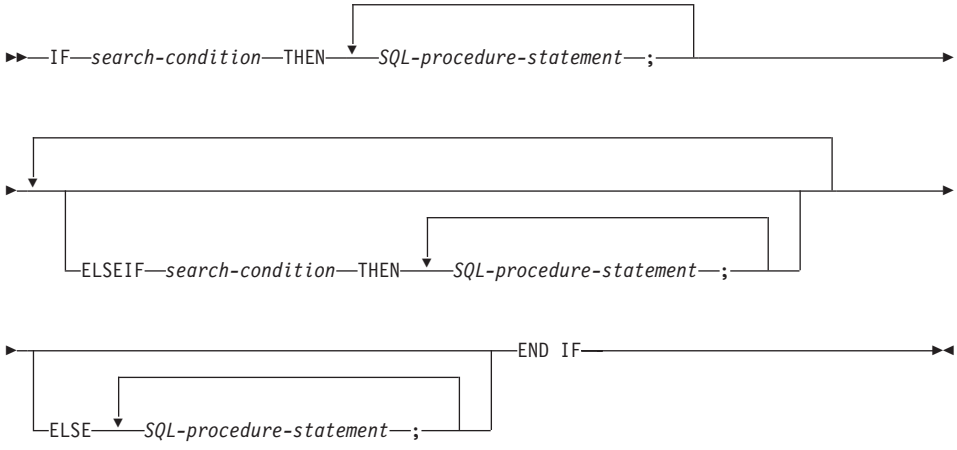
```
CREATE PROCEDURE adjust_salary
   (IN v_empno CHAR(6),
    IN rating INTEGER)
    OUT return_parm DECIMAL (8,2))
   MODIFIES SQL DATA
   LANGUAGE SQL
```

```
BEGIN
   DECLARE new_salary DECIMAL (9,2)
   DECLARE service DECIMAL (8,2)
     SELECT SALARY, CURRENT_DATE - HIREDATE
       INTO new_salary, service
       FROM EMPLOYEE
       WHERE EMPNO = v_empno
     IF service < 600
       THEN GOTO EXIT
     END IF
     IF rating = 1
       THEN SET new_salary = new_salary + (new_salary * .10)
     ELSE IF rating = 2
       THEN SET new_salary = new_salary + (new_salary * .05)
     END IF
     UPDATE EMPLOYEE
       SET SALARY = new_salary
       WHERE EMPNO = v_empno
     EXIT: SET return_parm = service
END
```

IF Statement

The IF statement selects an execution path based on the evaluation of a condition.

Syntax



Description

search-condition
Specifies the condition for which an SQL statement should be invoked. If the condition is unknown or false, processing continues to the next search condition, until either a condition is true or processing reaches the ELSE clause.

SQL-procedure-statement
Specifies the statement to be invoked if the preceding search-condition is true. If no search-condition evaluates to true, then the *SQL-procedure-statement* following the ELSE keyword is invoked.

Examples

The following SQL procedure accepts two IN parameters: an employee number *employee_number* and an employee rating *rating*. Depending on the value of *rating*, the `employee` table is updated with new values in the `salary` and `bonus` columns.

```
CREATE PROCEDURE UPDATE_SALARY_IF
   (IN employee_number CHAR(6), INOUT rating SMALLINT)
   LANGUAGE SQL
   BEGIN
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE EXIT HANDLER FOR not_found
         SET rating = -1;
      IF rating = 1
```

```
      THEN UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
      WHERE empno = employee_number;
   ELSEIF rating = 2
      THEN UPDATE employee
      SET salary = salary * 1.05, bonus = 500
      WHERE empno = employee_number;
   ELSE UPDATE employee
      SET salary = salary * 1.03, bonus = 0
      WHERE empno = employee_number;
   END IF;
END
```

ITERATE Statement

The ITERATE statement causes the flow of control to return to the beginning of a labelled loop.

Syntax

Description

label

Specifies the label of the FOR, LOOP, REPEAT, or WHILE statement to which DB2 passes the flow of control.

Examples

This example uses a cursor to return information for a new department. If the *not_found* condition handler was invoked, the flow of control passes out of the loop. If the value of *v_dept* is 'D11', an ITERATE statement passes the flow of control back to the top of the LOOP statement. Otherwise, a new row is inserted into the DEPARTMENT table.

```
CREATE PROCEDURE ITERATOR()
   LANGUAGE SQL
   BEGIN
      DECLARE v_dept CHAR(3);
      DECLARE v_deptname VARCHAR(29);
      DECLARE v_admdept CHAR(3);
      DECLARE at_end INTEGER DEFAULT 0;
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE c1 CURSOR FOR
         SELECT deptno, deptname, admrdept
         FROM department
         ORDER BY deptno;
      DECLARE CONTINUE HANDLER FOR not_found
         SET at_end = 1;
      OPEN c1;
      ins_loop:
      LOOP
         FETCH c1 INTO v_dept, v_deptname, v_admdept;
         IF at_end = 1 THEN
            LEAVE ins_loop;
         ELSEIF v_dept = 'D11' THEN
            ITERATE ins_loop;
         END IF;
         INSERT INTO department (deptno, deptname, admrdept)
         VALUES ('NEW', v_deptname, v_admdept);
      END LOOP;
      CLOSE c1;
   END
```

LEAVE Statement

The LEAVE statement transfers program control out of a loop or a compound statement.

Syntax

Description

label

Specifies the label of the compound, FOR, LOOP, REPEAT, or WHILE statement to exit.

Rules

When a LEAVE statement transfers control out of a compound statement, all open cursors in the compound statement, except cursors that are used to return result sets, are closed.

Examples

This example contains a loop that fetches data for cursor *c1*. If the value of SQL variable *at_end* is not zero, the LEAVE statement transfers control out of the loop.

```
CREATE PROCEDURE LEAVE_LOOP(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE at_end SMALLINT DEFAULT 0;
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
            FROM employee;
      DECLARE CONTINUE HANDLER for not_found
         SET at_end = 1;
      SET v_counter = 0;
      OPEN c1;
      fetch_loop:
      LOOP
         FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
         IF at_end <> 0 THEN LEAVE fetch_loop;
         END IF;
         SET v_counter = v_counter + 1;
      END LOOP fetch_loop;
      SET counter = v_counter;
      CLOSE c1;
   END
```

LOOP Statement

The LOOP statement repeats the execution of a statement or a group of statements.

Syntax

Description

label
Specifies the label for the LOOP statement. If the beginning label is specified, that label can be specified on LEAVE and ITERATE statements. If the ending label is specified, a matching beginning label must be specified.

SQL-procedure-statement
Specifies the statements to be invoked in the loop.

Examples

This procedure uses a LOOP statement to fetch values from the `employee` table. Each time the loop iterates, the OUT parameter *counter* is incremented and the value of *v_midinit* is checked to ensure that the value is not a single space (' '). If *v_midinit* is a single space, the LEAVE statement passes the flow of control outside of the loop.

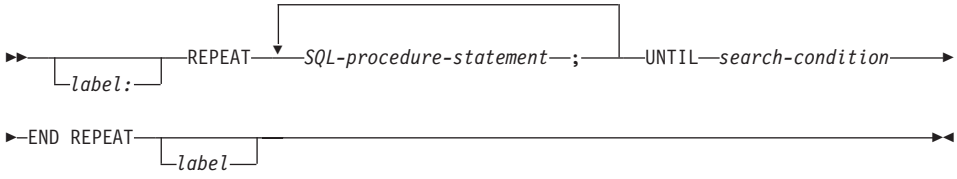
```
CREATE PROCEDURE LOOP_UNTIL_SPACE(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE v_firstnme VARCHAR(12);
      DECLARE v_midinit CHAR(1);
      DECLARE v_lastname VARCHAR(15);
      DECLARE c1 CURSOR FOR
         SELECT firstnme, midinit, lastname
            FROM employee;
      DECLARE CONTINUE HANDLER FOR NOT FOUND
         SET counter = -1;
      OPEN c1;
      fetch_loop:
      LOOP
         FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
         IF v_midinit = ' ' THEN
            LEAVE fetch_loop;
         END IF;
         SET v_counter = v_counter + 1;
```

```
   END LOOP fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

REPEAT Statement

The REPEAT statement executes a statement or group of statements until a search condition is true.

Syntax



Description

label
Specifies the label for the REPEAT statement. If the beginning label is specified, that label can be specified on LEAVE and ITERATE statements. If an ending label is specified, a matching beginning label also must be specified.

SQL-procedure-statement
Specifies the SQL statement to execute with the loop.

search-condition
Specifies a condition that is evaluated before each execution of the SQL procedure statement. If the condition is false, DB2 executes the SQL procedure statement in the loop.

Examples

A REPEAT statement fetches rows from a table until the *not_found* condition handler is invoked.

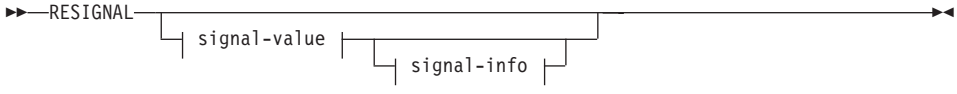
```
CREATE PROCEDURE REPEAT_STMT(OUT counter INTEGER)
   LANGUAGE SQL
   BEGIN
     DECLARE v_counter INTEGER DEFAULT 0;
     DECLARE v_firstnme VARCHAR(12);
     DECLARE v_midinit CHAR(1);
     DECLARE v_lastname VARCHAR(15);
     DECLARE at_end SMALLINT DEFAULT 0;
     DECLARE not_found CONDITION FOR SQLSTATE '02000';
     DECLARE c1 CURSOR FOR
        SELECT firstnme, midinit, lastname
           FROM employee;
     DECLARE CONTINUE HANDLER FOR not_found
        SET at_end = 1;
     OPEN c1;
     fetch_loop:
     REPEAT
```

```
      FETCH c1 INTO v_firstnme, v_midinit, v_lastname;
      SET v_counter = v_counter + 1;
      UNTIL at_end > 0
   END REPEAT fetch_loop;
   SET counter = v_counter;
   CLOSE c1;
END
```

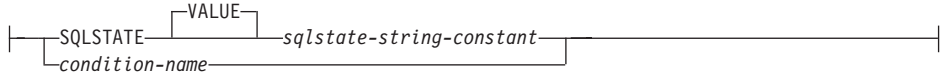
RESIGNAL Statement

The RESIGNAL statement is used to resignal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

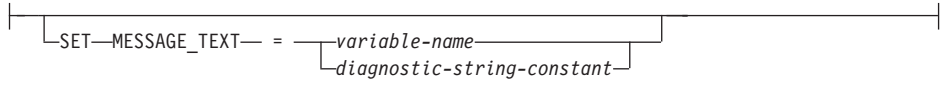
Syntax



signal-value:



signal-info:



Description

SQLSTATE VALUE *sqlstate-string-constant*
The specified string constant represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z')
- The SQLSTATE class (first two characters) cannot be '00', since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is raised (SQLSTATE 428B3).

condition-name
Specifies the name of the condition.

SET MESSAGE_TEXT =
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual string is

longer than 70 bytes, it is truncated without warning. This clause can only be specified if an SQLSTATE or *condition-name* is also specified (SQLSTATE 42601).

variable-name
Identifies an SQL variable that must be declared within the compound statement. The SQL variable must be defined as a CHAR or VARCHAR data type.

diagnostic-string-constant
Specifies a character string constant that contains the message text.

Notes

- If the RESIGNAL statement is specified without a SQLSTATE clause or a *condition-name*, the SQL routine returns to the caller with the identical condition that invoked the handler.
- If a RESIGNAL statement is issued, and specifies a SQLSTATE or *condition-name*, the SQLCODE assigned is:

  +438 if the SQLSTATE begins with '01' or '02'

  −438 otherwise

  When a RESIGNAL statement is issued with no options, the SQLCODE is unchanged from the exception that caused the handler to be invoked.
- If the SQLSTATE or condition indicates that an exception is signalled (an SQLSTATE class other than '01' or '02'):
  - then, the exception is handled and control is transferred to a handler, provided that a handler exists in the next outer compound statement (or a compound statement even further out) from the compound statement that includes the handler with the resignal statement, and the compound statement contains a handler for the specified SQLSTATE, condition-name, or SQLEXCEPTION;
  - otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.
- If the SQLSTATE or condition indicates that a warning (SQLSTATE class '01') or not found condition (SQLSTATE class '02') is signalled:
  - then the warning or not found condition is handled and control is transferred to a handler, provided that a handler exists in the next outer compound statement (or a compound statement even further out) from the compound statement that includes the handler with the resignal statement, and the compound statement contains a handler for the specified SQLSTATE, condition-name, SQLWARNING (if the SQLSTATE class is '01'), or NOT FOUND (if the SQLSTATE class is '02');
  - otherwise, the warning is not handled and processing continues with the next statement.

Examples

This example detects a division by zero error. The IF statement uses a SIGNAL statement to invoke the *overflow* condition handler. The condition handler uses a RESIGNAL statement to return a different SQLSTATE value to the client application.

```
CREATE PROCEDURE divide ( IN numerator INTEGER
                          IN denominator INTEGER
                          OUT result INTEGER
LANGUAGE SQL
CONTAINS SQL
BEGIN
  DECLARE overflow CONDITION FOR SQLSTATE '22003';
    DECLARE CONTINUE HANDLER FOR overflow
    RESIGNAL SQLSTATE'22375' ;
    IF denominator = 0 THEN
    SIGNAL overflow;
  ELSE
     SET result = numerator / denominator;
  END IF;
END
```

RETURN Statement

The RETURN statement is used to return from the routine. For SQL functions or methods, it returns the result of the function or method. For an SQL procedure, it optionally returns an integer status value.

Syntax

Description

expression
Specifies a value that is returned from the routine:

- If the routine is a function or method, *expression* must be specified (SQLSTATE 42630) and the data type of *expression* must be assignable to the RETURNS type of the routine (SQLSTATE 42866).
- If the routine is a procedure, the data type of *expression* must be INTEGER (SQLSTATE 428E2).

Notes

- When a value is returned from a procedure, the caller may access the value using:
  - the GET DIAGNOSTICS statement to retrieve the RETURN_STATUS when the SQL procedure was called from another SQL procedure
  - the parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI application
  - directly from the SQLCA returned from processing the CALL of an SQL procedure by retrieving the value of SQLERRD[0] when the SQLCODE is not less than zero (assume a value of −1 when SQLCODE is less than zero).

Examples

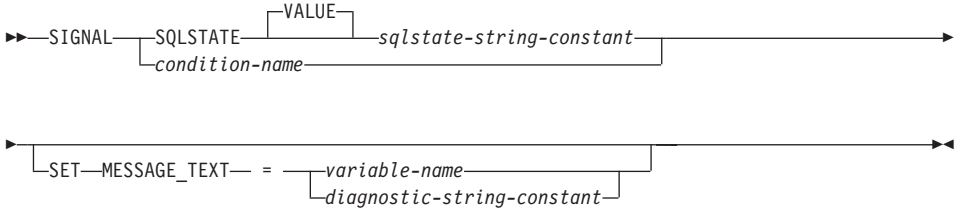
Use a RETURN statement to return from an SQL stored procedure with a status value of zero if successful, and −200 if not.

```
BEGIN
...
   GOTO FAIL
...
   SUCCESS: RETURN 0
   FAIL: RETURN -200
END
```

SIGNAL Statement

The SIGNAL statement is used to signal an error or warning condition. It causes an error or warning to be returned with the specified SQLSTATE, along with optional message text.

Syntax



Description

SQLSTATE VALUE sqlstate-string-constant
The specified string constant represents an SQLSTATE. It must be a character string constant with exactly 5 characters that follow the rules for SQLSTATEs:

- Each character must be from the set of digits ('0' through '9') or non-accented upper case letters ('A' through 'Z')
- The SQLSTATE class (first two characters) cannot be '00', since this represents successful completion.

If the SQLSTATE does not conform to these rules, an error is raised (SQLSTATE 428B3).

condition-name
Specifies the name of the condition. The condition name must be unique within the procedure and can only be referenced within the compound statement in which it is declared.

SET MESSAGE_TEXT=
Specifies a string that describes the error or warning. The string is returned in the SQLERRMC field of the SQLCA. If the actual string is longer than 70 bytes, it is truncated without warning. This clause can only be specified if a SQLSTATE or condition-name is also specified (SQLSTATE 42601).

variable-name
Identifies an SQL variable that must be declared within the compound statement. The SQL variable must be defined as a CHAR or VARCHAR data type.

diagnostic-string-constant
Specifies a character string constant that contains the message text.

Notes

- If a SIGNAL statement is issued, the SQLCODE that is assigned is:

  +438 if the SQLSTATE begins with '01' or '02'

  −438 otherwise
- If the SQLSTATE or condition indicates that an exception (SQLSTATE class other than ’01’ or ’02’) is signaled:
  - then the exception is handled and control is transferred to a handler, provided that a handler exists in the same compound statement (or an outer compound statement) as the signal statement, and the compound statement contains a handler for the specified SQLSTATE, condition-name, or SQLEXCEPTION;
  - otherwise, the exception is not handled and control is immediately returned to the end of the compound statement.
- If the SQLSTATE or condition indicates that a warning (SQLSTATE class ’01’) or not found condition (SQLSTATE class ’02’) is signaled:
  - then the warning or not found condition is handled and control is transferred to a handler, provided that a handler exists in the same compound statement (or an outer compound statement) as the signal statement, and the compound statement contains a handler for the specified SQLSTATE, condition-name, SQLWARNING (if the SQLSTATE class is ’01’), or NOT FOUND (if the SQLSTATE class is ’02’);
  - otherwise, the warning is not handled and processing continues with the next statement.
- SQLSTATE values are comprised of a two-character class code value, followed by a three-character subclass code value. Class code values represent classes of successful and unsuccessful execution conditions.

  Any valid SQLSTATE value can be used in the SIGNAL statement. However, it is recommended that programmers define new SQLSTATEs based on ranges reserved for applications. This prevents the unintentional use of an SQLSTATE value that might be defined by the database manager in a future release.
  - SQLSTATE classes that begin with the characters '7' through '9', or 'I' through 'Z' may be defined. Within these classes, any subclass may be defined.
  - SQLSTATE classes that begin with the characters '0' through '6', or 'A' through 'H' are reserved for the database manager. Within these classes, subclasses that begin with the characters '0' through 'H' are reserved for the database manager. Subclasses that begin with the characters 'I' through 'Z' may be defined.

Examples

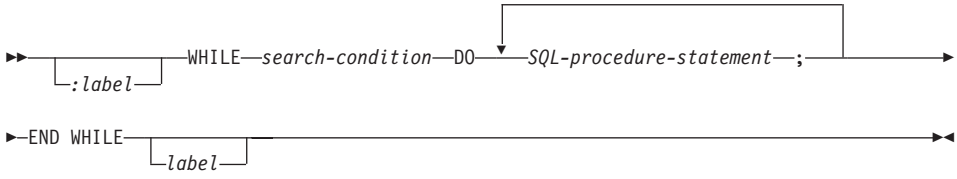
An SQL procedure for an order system that signals an application error when a customer number is not known to the application. The ORDERS table includes a foreign key to the CUSTOMER table, requiring that the CUSTNO exist before an order can be inserted.

```
CREATE PROCEDURE SUBMIT_ORDER
   (IN ONUM INTEGER, IN CNUM INTEGER,
    IN PNUM INTEGER, IN QNUM INTEGER)
    SPECIFIC SUBMIT_ORDER
    MODIFIES SQL DATA
   LANGUAGE SQL
   BEGIN
      DECLARE EXIT HANDLER FOR SQLSTATE VALUE '23503'
         SIGNAL SQLSTATE '75002'
            SET MESSAGE_TEXT = 'Customer number is not known';
      INSERT INTO ORDERS (ORDERNO, CUSTNO, PARTNO, QUANTITY)
         VALUES (ONUM, CNUM, PNUM, QNUM);
    END
```

WHILE Statement

The WHILE statement repeats the execution of a statement or group of statements while a specified condition is true.

Syntax



Description

label
Specifies the label for the WHILE statement. If the beginning label is specified, it can be specified in LEAVE and ITERATE statements. If the ending label is specified, it must be the same as the beginning label.

search-condition
Specifies a condition that is evaluated before each execution of the loop. If the condition is true, the SQL-procedure-statements in the loop are processed.

SQL-procedure-statement
Specifies the SQL statement or statements to execute within the loop.

Examples

This example uses a WHILE statement to iterate through FETCH and SET statements. While the value of SQL variable *v_counter* is less than half of number of employees in the department identified by the IN parameter *deptNumber*, the WHILE statement continues to perform the FETCH and SET statements. When the condition is no longer true, the flow of control leaves the WHILE statement and closes the cursor.

```
CREATE PROCEDURE DEPT_MEDIAN
   (IN deptNumber SMALLINT, OUT medianSalary DOUBLE)
   LANGUAGE SQL
   BEGIN
      DECLARE v_numRecords INTEGER DEFAULT 1;
      DECLARE v_counter INTEGER DEFAULT 0;
      DECLARE c1 CURSOR FOR
         SELECT CAST(salary AS DOUBLE)
            FROM staff
            WHERE DEPT = deptNumber
            ORDER BY salary;
      DECLARE EXIT HANDLER FOR NOT FOUND
         SET medianSalary = 6666;
      SET medianSalary = 0;
```

```
   SELECT COUNT(*) INTO v_numRecords
         FROM staff
         WHERE DEPT = deptNumber;
   OPEN c1;
   WHILE v_counter < (v_numRecords / 2 + 1) DO
      FETCH c1 INTO medianSalary;
      SET v_counter = v_counter + 1;
   END WHILE;
   CLOSE c1;
END
```