PL/SQL subprograms

Subprograms are named PL/SQL blocks that can take parameters and be invoked. PL/SQL has two types of subprograms called procedures and functions. Generally, you use a procedure to perform an action and a function to compute a value.

Like unnamed or anonymous PL/SQL blocks, subprograms have a declarative part, an executable part, and an optional exception-handling part. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local and cease to exist when you exit the subprogram. The executable part contains statements that assign values, control execution, and manipulate Oracle data. The exception-handling part contains exception handlers, which deal with exceptions raised during execution.

Understanding PL/SQL Procedures

A procedure is a subprogram that performs a specific action. You write procedures using the syntax:

```sql
[CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
   [AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
   [PRAGMA AUTONOMOUS_TRANSACTION;]
   [local declarations]
BEGIN
   executable statements
   [EXCEPTION
   exception handlers]
END [name];
```

where parameter stands for the following syntax:

```sql
parameter_name [IN | OUT [NOCOPY] | IN OUT [NOCOPY]] datatype
   [{:= | DEFAULT} expression]
```

The AUTHID clause determines whether a stored procedure executes with the privileges of its owner (the default) or current user and whether its unqualified references to schema objects are resolved in the schema of the owner or current user. You can override the default behavior by specifying CURRENT_USER.

The pragma AUTONOMOUS_TRANSACTION instructs the PL/SQL compiler to mark a procedure as autonomous (independent). Autonomous transactions let you suspend the main transaction, do SQL operations, commit or roll back those operations, then resume the main transaction.
You cannot constrain the datatype of a parameter. For example, the following
declaration of `acct_id` is illegal because the datatype `CHAR` is size-constrained:

```
PROCEDURE reconcile (acct_id CHAR(5)) IS ...  -- illegal
```

However, you can use the following workaround to size-constrain parameter types indirectly:

```
DECLARE
    SUBTYPE Char5 IS CHAR(5);
    PROCEDURE reconcile (acct_id Char5) IS ...
```

A procedure has two parts: the `specification` (spec for short) and the `body`. The procedure spec begins with the keyword `PROCEDURE` and ends with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses.

Consider the procedure `raise_salary`, which increases the salary of an employee by a given amount:

```
PROCEDURE raise_salary (emp_id INTEGER, amount REAL) IS
    current_salary REAL;
    salary_missing EXCEPTION;
BEGIN
    SELECT sal INTO current_salary FROM emp
        WHERE empno = emp_id;
    IF current_salary IS NULL THEN
        RAISE salary_missing;
    ELSE
        UPDATE emp SET sal = sal + amount
            WHERE empno = emp_id;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO emp_audit VALUES (emp_id, 'No such number');
    WHEN salary_missing THEN
        INSERT INTO emp_audit VALUES (emp_id, 'Salary is null');
END raise_salary;
```

Understanding PL/SQL Functions

```
[CREATE  [OR REPLACE ] ]
FUNCTION function_name [ ( parameter [ , parameter ]... ) ] RETURN
datatype
    [ AUTHID { DEFINER | CURRENT_USER } ]
    [DETERMINISTIC]
{IS | AS}
    [ PRAGMA AUTONOMOUS_TRANSACTION; ]
    [ local declarations ]
BEGIN
    executable statements
    [ EXCEPTION
        exception handlers ]
END [ name ];
```
The hint `DETERMINISTIC` helps the optimizer avoid redundant function calls. If a stored function was called previously with the same arguments, the optimizer can elect to use the previous result. The function result should not depend on the state of session variables or schema objects. Otherwise, results might vary across calls.

You cannot constrain (with `NOT NULL` for example) the datatype of a parameter or a function return value. However, you can use a workaround to size-constrain them indirectly.

Consider the function `sal_ok`, which determines if a salary is out of range:

```plsql
FUNCTION sal_ok (salary REAL, title VARCHAR2) RETURN BOOLEAN IS
    min_sal REAL;
    max_sal REAL;
BEGIN
    SELECT losal, hisal INTO min_sal, max_sal FROM sals
    WHERE job = title;
    RETURN (salary >= min_sal) AND (salary <= max_sal);
END sal_ok;
```

Controlling Side Effects of PL/SQL Subprograms

To be callable from SQL statements, a stored function must obey the following "purity" rules, which are meant to control side effects:

- When called from a `SELECT` statement or a parallelized `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot modify any database tables.
- When called from an `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot query or modify any database tables modified by that statement.
- When called from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute SQL transaction control statements (such as `COMMIT`), session control statements (such as `SET ROLE`), or system control statements (such as `ALTER SYSTEM`). Also, it cannot execute DDL statements (such as `CREATE`) because they are followed by an automatic commit.

If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed).

Declaring PL/SQL Subprograms

You can declare subprograms in any PL/SQL block, subprogram, or package. But, you must declare subprograms at the end of a declarative section after all other program items.

PL/SQL requires that you declare an identifier before using it. Therefore, you must declare a subprogram before calling it. For example, the following declaration of
procedure award_bonus is illegal because award_bonus calls procedure calc_rating, which is not yet declared when the call is made:

DECLARE
  ...
  PROCEDURE award_bonus IS
  BEGIN
    calc_rating(...);  -- undeclared identifier
  ...
  ...
  PROCEDURE calc_rating (...) IS
  BEGIN
    ...
  END;

In this case, you can solve the problem easily by placing procedure calc_rating before procedure award_bonus. However, the easy solution does not always work. For example, suppose the procedures are mutually recursive (call each other) or you want to define them in logical or alphabetical order.

You can solve the problem by using a special subprogram declaration called a forward declaration, which consists of a subprogram spec terminated by a semicolon. In the following example, the forward declaration advises PL/SQL that the body of procedure calc_rating can be found later in the block.

DECLARE
  PROCEDURE calc_rating ( ... );  -- forward declaration
  ...

Positional Versus Named Notation for Subprogram Parameters

When calling a subprogram, you can write the actual parameters using either positional or named notation. That is, you can indicate the association between an actual and formal parameter by position or name. So, given the declarations

DECLARE
  acct INTEGER;
  amt REAL;
  PROCEDURE credit_acct (acct_no INTEGER, amount REAL) IS ...

you can call the procedure credit_acct in four logically equivalent ways:

BEGIN
  credit_acct(acct, amt);  -- positional notation
  credit_acct(amount => amt, acct_no => acct);  -- named notation
  credit_acct(acct_no => acct, amount => amt);  -- named notation
  credit_acct(acct, amount => amt);  -- mixed notation
Specifying Subprogram Parameter Modes

You use parameter modes to define the behavior of formal parameters. The three parameter modes, \texttt{IN} (the default), \texttt{OUT}, and \texttt{IN OUT}, can be used with any subprogram. However, avoid using the \texttt{OUT} and \texttt{IN OUT} modes with functions. The purpose of a function is to take zero or more arguments (actual parameters) and return a single value. To have a function return multiple values is a poor programming practice. Also, functions should be free from \textit{side effects}, which change the values of variables not local to the subprogram.

Using the \texttt{IN} Mode

\texttt{An IN} parameter lets you pass values to the subprogram being called. Inside the subprogram, an \texttt{IN} parameter acts like a constant. Therefore, it cannot be assigned a value. For example, the following assignment statement causes a compilation error:

\begin{verbatim}
PROCEDURE debit_account (acct_id IN INTEGER, amount IN REAL) IS
  minimum_purchase CONSTANT REAL DEFAULT 10.0;
  service_charge   CONSTANT REAL DEFAULT 0.50;
BEGIN
  IF amount < minimum_purchase THEN
    amount := amount + service_charge; -- causes compilation error
  END IF;
  ...
END debit_account;
\end{verbatim}

Using the \texttt{OUT} Mode

\texttt{An OUT} parameter lets you return values to the caller of a subprogram. Inside the subprogram, an \texttt{OUT} parameter acts like a variable. That means you can use an \texttt{OUT} formal parameter as if it were a local variable. You can change its value or reference the value in any way, as the following example shows:

\begin{verbatim}
PROCEDURE calc_bonus (emp_id IN INTEGER, bonus OUT REAL) IS
  hire_date     DATE;
  bonus_missing EXCEPTION;
BEGIN
  SELECT sal * 0.10, hiredate INTO bonus, hire_date FROM emp
    WHERE empno = emp_id;
  IF bonus IS NULL THEN
    RAISE bonus_missing;
  END IF;
  IF MONTHS_BETWEEN(SYSDATE, hire_date) > 60 THEN
    bonus := bonus + 500;
  END IF;
  ...
EXCEPTION
  WHEN bonus_missing THEN
    ...
END calc_bonus;
\end{verbatim}
Using the IN OUT Mode

An IN OUT parameter lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, an IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable.

The actual parameter that corresponds to an IN OUT formal parameter must be a variable; it cannot be a constant or an expression.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

Summary of Subprogram Parameter Modes

<table>
<thead>
<tr>
<th>IN</th>
<th>OUT</th>
<th>IN OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>the default</td>
<td>must be specified</td>
<td>must be specified</td>
</tr>
<tr>
<td>passes values to a subprogram</td>
<td>returns values to the caller</td>
<td>passes initial values to a subprogram and returns updated values to the caller</td>
</tr>
<tr>
<td>formal parameter acts like a constant</td>
<td>formal parameter acts like a variable</td>
<td>formal parameter acts like an initialized variable</td>
</tr>
<tr>
<td>formal parameter cannot be assigned a value</td>
<td>formal parameter must be assigned a value</td>
<td>formal parameter should be assigned a value</td>
</tr>
<tr>
<td>actual parameter can be a constant, initialized variable, literal, or expression</td>
<td>actual parameter must be a variable</td>
<td>actual parameter must be a variable</td>
</tr>
<tr>
<td>actual parameter is passed by reference (a pointer to the value is passed in)</td>
<td>actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified</td>
<td>actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified</td>
</tr>
</tbody>
</table>

Using Default Values for Subprogram Parameters

As the example below shows, you can initialize IN parameters to default values. That way, you can pass different numbers of actual parameters to a subprogram, accepting or overriding the default values as you please. Moreover, you can add new formal parameters without having to change every call to the subprogram.
PROCEDURE create_dept (  
    new_dname VARCHAR2 DEFAULT 'TEMP',  
    new_loc   VARCHAR2 DEFAULT 'TEMP') IS  
BEGIN  
    INSERT INTO dept  
      VALUES (deptno_seq.NEXTVAL, new_dname, new_loc);  
...  
END;  

If an actual parameter is not passed, the default value of its corresponding formal parameter is used. Consider the following calls to create_dept:

create_dept;
create_dept('MARKETING');
create_dept('MARKETING', 'NEW YORK');

The first call passes no actual parameters, so both default values are used. The second call passes one actual parameter, so the default value for new_loc is used. The third call passes two actual parameters, so neither default value is used.

However, you cannot skip a formal parameter by leaving out its actual parameter. For example, the following call incorrectly associates the actual parameter 'NEW YORK' with the formal parameter new_dname:

create_dept('NEW YORK');  -- incorrect

In such cases, you must use named notation, as follows:

create_dept(new_loc => 'NEW YORK');

Overloading Subprogram Names

PL/SQL lets you overload subprogram names and type methods. That is, you can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

Suppose you want to initialize the first \( n \) rows in two index-by tables that were declared as follows:

DECLARE  
    TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;  
    TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;  
    hiredate_tab DateTabTyp;  
    sal_tab RealTabTyp;  
BEGIN  
    ...  
END;
You might write the following procedure to initialize the index-by table named `hiredate_tab`:

```sql
PROCEDURE initialize (tab OUT DateTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
   END LOOP;
END initialize;
```

And, you might write the next procedure to initialize the index-by table named `sal_tab`:

```sql
PROCEDURE initialize (tab OUT RealTabTyp, n INTEGER) IS
BEGIN
   FOR i IN 1..n LOOP
      tab(i) := 0.0;
   END LOOP;
END initialize;
```

Because the processing in these two procedures is the same, it is logical to give them the same name.

You can place the two overloaded `initialize` procedures in the same block, subprogram, or package. PL/SQL determines which of the two procedures is being called by checking their formal parameters.

**Restrictions on Overloading**

Only local or packaged subprograms, or type methods, can be overloaded. Therefore, you cannot overload standalone subprograms. Also, you cannot overload two subprograms if their formal parameters differ only in name or parameter mode. For example, you cannot overload the following two procedures:

```sql
DECLARE
   ...
   PROCEDURE reconcile (acct_no IN INTEGER) IS
      BEGIN ... END;
   PROCEDURE reconcile (acct_no OUT INTEGER) IS
      BEGIN ... END;
```

You cannot overload two subprograms if their formal parameters differ only in datatype and the different datatypes are in the same family. For instance, you cannot overload the following procedures because the datatypes `INTEGER` and `REAL` are in the same family:
DECLARE
...
PROCEDURE charge_back (amount INTEGER) IS
BEGIN ... END;
PROCEDURE charge_back (amount REAL) IS
BEGIN ... END;

Likewise, you cannot overload two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family. For example, you cannot overload the following procedures because the base types CHAR and LONG are in the same family:

DECLARE
  SUBTYPE Delimiter IS CHAR;
  SUBTYPE Text IS LONG;
...
PROCEDURE scan (x Delimiter) IS
BEGIN ... END;
PROCEDURE scan (x Text) IS
BEGIN ... END;

Finally, you cannot overload two functions that differ only in return type (the datatype of the return value) even if the types are in different families. For example, you cannot overload the following functions:

DECLARE
...
FUNCTION acct_ok (acct_id INTEGER) RETURN BOOLEAN IS
BEGIN ... END;
FUNCTION acct_ok (acct_id INTEGER) RETURN INTEGER IS
BEGIN ... END;

Invoker Rights Versus Definer Rights

By default, stored procedures and SQL methods execute with the privileges of their owner, not their current user. Such definer-rights subprograms are bound to the schema in which they reside. For example, assume that dept tables reside in schemas scott and blake, and that the following standalone procedure resides in schema scott:

CREATE PROCEDURE create_dept (
  my_deptno NUMBER,
  my_dname VARCHAR2,
  my_loc VARCHAR2) AS
BEGIN
  INSERT INTO dept VALUES (my_deptno, my_dname, my_loc);
END;

Also assume that user scott has granted the EXECUTE privilege on this procedure to user blake. When user blake calls the procedure, the INSERT statement executes...
with the privileges of user scott. Also, the unqualified references to table dept is resolved in schema scott. So, even though user blake called the procedure, it updates the dept table in schema scott.

On the other hand, invoker-rights subprograms are not bound to a particular schema. They can be run by a variety of users. The following version of procedure create_dept executes with the privileges of its current user and inserts rows into the dept table in that user's schema:

```sql
CREATE PROCEDURE create_dept (my_deptno NUMBER, my_dname VARCHAR2, my_loc VARCHAR2) AUTHID CURRENT_USER AS
BEGIN
    INSERT INTO dept VALUES (my_deptno, my_dname, my_loc);
END;
```

Understanding and Using Recursion

A recursive subprogram is one that calls itself. Think of a recursive call as a call to some other subprogram that does the same task as your subprogram. Each recursive call creates a new instance of any items declared in the subprogram, including parameters, variables, cursors, and exceptions. Likewise, new instances of SQL statements are created at each level in the recursive descent.

Be careful where you place a recursive call. If you place it inside a cursor FOR loop or between OPEN and CLOSE statements, another cursor is opened at each call. As a result, your program might exceed the limit set by the Oracle initialization parameter OPEN_CURSORS.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a terminating condition. Otherwise, the recursion would (theoretically) go on forever. In practice, if a recursive subprogram strays into infinite regress, PL/SQL eventually runs out of memory and raises the predefined exception STORAGE_ERROR.

The following recursive function returns the factorial of a positive integer:

```sql
FUNCTION fac (n POSITIVE) RETURN INTEGER IS -- returns n!
BEGIN
    IF n = 1 THEN -- terminating condition
        RETURN 1;
    ELSE
        RETURN n * fac(n - 1); -- recursive call
    END IF;
END fac;
```
Recursion Example: Traversing Tree-Structured Data

Consider the procedure below, which finds the staff of a given manager. The procedure declares two formal parameters, \( \text{mgr\_no} \) and \( \text{tier} \), which represent the manager's employee number and a tier in his or her departmental organization. Staff members reporting directly to the manager occupy the first tier.

```sql
PROCEDURE find_staff (mgr_no NUMBER, tier NUMBER := 1) IS
    boss_name VARCHAR2(10);
    CURSOR c1 (boss_no NUMBER) IS
        SELECT empno, ename FROM emp WHERE mgr = boss_no;
    BEGIN
        /* Get manager's name. */
        SELECT ename INTO boss_name FROM emp WHERE empno = mgr_no;
        IF tier = 1 THEN
            INSERT INTO staff -- single-column output table
            VALUES (boss_name || ' manages the staff');
        END IF;
        /* Find staff members who report directly to manager. */
        FOR ee IN c1 (mgr_no) LOOP
            INSERT INTO staff
            VALUES (boss_name || ' manages ' || ee.ename
                    || ' on tier ' || TO_CHAR(tier));
            /* Drop to next tier in organization. */
            find_staff(ee.empno, tier + 1); -- recursive call
        END LOOP;
    COMMIT;
END;
```

Tip: Perform Recursive Queries with the CONNECT BY Clause

The last example illustrates recursion, not the efficient use of set-oriented SQL statements. You might want to compare the performance of the recursive procedure to that of the following SQL statement, which does the same task:

```sql
INSERT INTO staff
    SELECT PRIOR ename || ' manages ' || ename
        || ' on tier ' || TO_CHAR(LEVEL - 1)
    FROM emp
START WITH empno = 7839
CONNECT BY PRIOR empno = mgr;
```