

**MP0484. Bases de datos
UF7. Programación avanzada
de acceso a datos**

7.1. Procedimientos y Funciones Almacenados

Índice

☰	Objetivos	3
☰	Introducción a Bloques Almacenados	4
☰	Levantar excepciones especiales y su tratamiento	7
☰	Procedimientos Almacenados	10
☰	Funciones Almacenadas	21

Objetivos

Procedimientos y Funciones, son bloques almacenados (creados) en mi usuario, y que puedo llamar desde cualquier bloque anónimo o almacenado para realizar acciones específicas con mis tablas

Objetivos:

- Entender la sintaxis de una bloque almacenado.
- Conocer los tipos de parámetros que recibe un bloque almacenado.
- Entender la diferencia entre parámetros In y OUT.
- Entender la relación de confirmación de actualizaciones entre un bloque llamante y un bloque llamado.
- Comprender cómo se ejecuta una función y en qué tipo de sentencias PL/SQL se pueden incorporar.

Introducción a Bloques Almacenados

Un código PL / SQL se dice que está **almacenado**, porque al compilarle, el motor de Oracle lo guarda como un componente más de la Base de Datos, y por tanto se puede solicitar su ejecución desde otro PL / SQL, o desde otros entornos, siempre y cuando se haya establecido la conexión a nuestra base de datos, tenga los permisos oportunos y esté compilado correctamente.

El mandato SQL/DDL empleado para su generación, es **CREATE**.

Los bloques almacenados que se pueden crear son:

1. Procedimientos : **CREATE PROCEDURE**
2. Funciones: **CREATE FUNCTION**
3. Paquetes: **CREATE PACKAGE**
4. Disparadores: **CREATE TRIGGER**

Los bloques almacenados se compilan, y tengan o no tengan errores de compilación se guardan en el conjunto de objetos de mi usuario, pero:

- Si tienen errores de compilación su **STATUS** estará **INVALID**, y no se le puede llamar a ejecutar.
- Si no tiene errores de compilación su **STATUS** será **VALID**, y se le puede llamar a ejecutar(siempre que tengas permiso para ello).
- La columna **STATUS**, se encuentra en la Vista del diccionario **USER_OBJECTS / ALL_OBJECTS**.

Procedimientos y Funciones almacenados son bloques PL / SQL con un *nombre*, a los que se pueden pasar *parámetros* y pueden ser invocados desde otros bloques, tanto anónimos como almacenados. Estos bloques se guardan en la base de datos una vez creados, sin(o con) errores de compilación.

PL / SQL distingue entre los dos tipos de subprogramas:

- los procedimientos, que se utilizan para ejecutar una acción;
- y las funciones, que siempre retornan un valor.

Las partes fundamentales de un subprograma son:

- **Las especificaciones:** donde se define el tipo del subprograma (procedimiento o función), el nombre del mismo y los parámetros de entrada y / o salida. Los parámetros son opcionales.
- **Una parte declarativa** (No se usa la cláusula DECLARE): donde se definen todos los elementos que forman parte del subprograma: variables, constantes, cursores, tipos de datos, subprogramas, etc.
- **Una parte de Ejecución:** que comienza con BEGIN donde se realizan todas las acciones, sentencias de control y sentencia SQL.
- **Una parte de excepciones** o de control de errores (opcional), se especifique las acciones a tomar en caso de que se produzca un error en la ejecución del subprograma.

Utilidad de procedimientos y Funciones almacenados

Como sabemos de SQL, todas las tablas que crea un usuario, son propiedad de ese usuario, y sólo de él.

Si quiero que otros usuarios puedan consultar, y/o actualizar información de cualquiera de mis tablas, les tengo que dar permisos, por ejemplo:

```
GRANT SELECT, INSERT, UPDATE ON EMPLOYEES TO ALUMNO;
```

Autorizo al usuario ALUMNO a leer, insertar y modificar en mi tabla employees.

Pero si no quiero (o no me fío del conocimiento que tenga el usuario ALUMNO de SQL), tengo la posibilidad de crear Bloques PL/SQL almacenados, es decir, procedimientos y funciones, para encerrar ahí las sentencias SQL, y le doy autorización a ejecutar estos bloques, desde PL o desde cualquier lenguaje de programación (como por ejemplo java):

```
GRANT EXECUTE ON NOMBRE_PROCEDIMENTO TO ALUMNO;
```

```
GRANT EXECUTE ON NOMBRE_FUNCION TO ALUMNO;
```

Levantar excepciones especiales y su tratamiento

Los bloques anónimos procedimientos, Funciones y disparadores, cuando levantan excepciones, lo hacen a través del el procedimiento RAISE_APPLICATION_ERROR que permite, no solo levantar la excepción, sino definir mensajes de error del tipo ORA- por el usuario. La sintaxis es:

```
RAISE_APPLICATION_ERROR (numero_error, mensaje_error);
```

El numero_error tiene el rango de -20000 a -20999 y mensaje puede tener una longitud máxima de 2048 bytes.

RAISE_APPLICATION_ERROR está definido en el paquete DBMS_STANDARD por lo que se puede invocar desde cualquier programa o subprograma PL/SQL almacenado.

Cuando RAISE_APPLICATION_ERROR es invocado el subprograma acaba y devuelve el número de error y el mensaje a la aplicación o bloque que le invocó.

Tratamiento por el bloque llamante

El bloque que llama a otro que le levanta una excepción de tipo RAISE_APPPLICATION_ERROR(-20XXX, 'mensaje_error'), tiene dos formas de tratarle:

1

De forma genérica, capturándole a través del WHEN OTHERS THEN.

```

.....
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERROR GENERAL : ' || SQLERRM);
END;
/

-- CASO DE RECIBIR UN ERROR LA SALIDA SERIA
'ERROR GENERAL : -20XXX MENSAJE ERROR CORRESPONDIENTE

```

2

De forma específica, capturando a través de un nombre de excepción.

Supongamos que un bloque almacenado detecta que algo no va bien y nos manda un código Oracle -20100, con el mensaje 'error de proceso'. Y queremos capturarle a través de un tratamiento específico.

Procederemos de la siguiente forma:

```

DECLARE
    proceso_erroneo          EXCEPTION;
    .....
    PRAGMA EXCEPTION_INIT(proceso_erroneo, -20100);
BEGIN
    procedimiento1;

EXCEPTION
    WHEN proceso_erroneo THEN
        DBMS_OUTPUT.PUT_LINE('ERROR: ' || SQLERRM);
END;
/

```

1. Declaramos la excepción proceso_erroneo.
2. Declaramos un PRAGMA, directiva para oracle, EXCEPTION_INIT, asignando el código -20100 a mi excepción.
3. Llamo al procedimiento "procedimiento1".
4. Este me levanta un código -20100, y el sistema se lo asigna a mi excepción.
5. La capturo: when proceso_erroneo Then...
6. Y pongo las instrucciones necesarias para tratar la excepción.

Procedimientos Almacenados

Definición formal

La sintaxis para la definición de un procedimiento es:

```
CREATE [OR REPLACE]
PROCEDURE Nombre_procedimiento [(declaración de parámetros)]
    [AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [Declaraciones locales de tipos, variables, etc]
BEGIN
    Sentencias ejecutables del procedimiento
[EXCEPTION
    Excepciones definidas y las acciones de estas excepciones]
END [Nombre_procedimiento];
/
```

CREATE OR REPLACE

Permite crear un procedimiento STANDALONE (No forma parte de un paquete) y guardarlo dentro de la base de datos. Si se crea un procedimiento que ya existe dará error por ello se utiliza la cláusula OR REPLACE. Si el procedimiento no existe se creará y si ya existe se remplazará.

AUTHID

La cláusula AUTHID determina si un procedimiento se ejecuta con los privilegios del usuario que lo ha creado (por defecto) o si con los privilegios del usuario que lo invoca. También determina si las referencias no cualificadas las resuelve en el esquema del propietario del procedimiento o de quién lo invoca. Para especificar que se ejecute con los permisos de quién lo invoca se utiliza la cláusula CURRENT_USER.

Pragma AUTONOMOUS_TRANSACTION

Marca el procedimiento como autónomo. Un procedimiento autónomo permite realizar COMMIT o ROLLBACK de las sentencias SQL propias sin afectar a la transacción que lo haya llamado. Si en ejecución no se encuentra la sentencia COMMIT o ROLBAK provocará una excepción

NO DECLARE

A diferencia de los Bloques Anónimos, **NO** se usa la cláusula DECLARE puesto que va implícita en el IS o el AS.

No existe diferencia en utilizar el IS o el AS.

Este es el procedimiento más pequeño que se puede hacer sin que de errores de compilación, eso sí no hace nada:

```
CREATE OR REPLACE PROCEDURE no_HACE_NADA AS
BEGIN
    NULL;
END IMP_LIN;
```

Parámetros

Un procedimiento puede o no tener parámetros de entrada, si no tiene parámetros no es necesario los paréntesis, ni en la cabecera del procedimiento ni en la llamada al procedimiento.

Para definir parámetros, se especifica el nombre de la variable y a continuación el TIPO de parámetro ORACLE y/o PL/SQL, **sin precisión**:

```
DATE, VARCHAR2, CHAR, DECIMAL , nombre_tabla.nombre_columna%type,
nombre_tabla%ROWTYPE... .
```

- entre paréntesis y separados por comas.

Ejemplo: vamos a crear un procedimiento en mi usuario(recordad que estamos con el usuario HR), que imprima por consola una línea de separación, para lo cual me hace falta el carácter que forma parte de la línea, y cuantos caracteres la conforman. Le voy a llamar IMP_LIN.

```
CREATE OR REPLACE PROCEDURE IMP_LIN(p_car char, p_cuantos pls_integer) AS
BEGIN
    dbms_output.put_line(RPAD(p_car, p_cuantos, p_car));
END IMP_LIN;
```

El tipo char sin precisión.

Parámetros separados por comas.

No hay cláusula DECLARE.

```
CREATE OR REPLACE PROCEDURE IMP_LIN(p_car char, p_cuantos pls_integer) AS
BEGIN
    dbms_output.put_line(RPAD(p_car, p_cuantos, p_car));
END IMP_LIN;
```

Poner el nombre del procedimiento detrás del END es una buena práctica.

Las variables se usan dentro del procedimiento.

Esquema de procedimiento

Para probar que el procedimiento funciona creamos un bloque anónimo, con distintas llamadas:

```

set serveroutput on

begin

    imp_lin('=',60);
    imp_lin('--FIN',40);
    imp_lin('*',20);
end;

/
=====
--FIN--FIN--FIN--FIN--FIN--FIN--FIN--FIN
*****
Procedimiento PL/SQL terminado correctamente.

```

Llamada a IMP_LIN

La ventaja que los parámetros sean sin precisión es que le puedo poner al parámetro todos los caracteres que quiera.

Tipos de parámetros

Los parámetros, entre el nombre y el tipo de datos llevan un modificador, y pueden ser de tres tipos:

- IN
- OUT
- IN OUT

IN

Parámetros de entrada. Suministra valores al procedimiento y el valor se trata como si fuera una **constante**. No se puede modificar en el valor dentro del procedimiento.

Se puede especificar un valor por defecto por si no es suministrado al invocar el subprograma.

Es la opción por defecto.

- Cuando invoco a un procedimiento, a un parámetro IN le puedo pasar una variable, o un literal del tipo correspondiente.

OUT

- A un parámetro OUT, en la invocación, sólo puedo pasar una variable, nunca un literal.
- Parámetros de salida. PL crea dentro del procedimiento una copia de esta variable pasada y la inicializa a NULL.
- Si el procedimiento termina bien, el contenido de esta variable, se copia en la variable del bloque que invocó al procedimiento
- Si el bloque termina levantando una excepción de tipo RAISE_APPLICATION_ERROR(-20XXX,'mensaje de error'), el contenido no se copia en la variable de origen.
- Si el bloque termina levantando una excepción de tipo RAISE_APPLICATION_ERROR(-20XXX,'mensaje de error'), y hemos especificado la opción OUT NOCOPY, el procedimiento trabaja directamente con la variable de origen, y cualquier cambio que hagamos dentro del procedimiento afecta a la variable pasada. Termine el programa bien o mal, el contenido queda cambiado.

IN OUT

- Parámetro de entrada y Salida. Se inicializa en el momento de invocar el subprograma y se trata como una variable dentro de él. Se comporta con las ventajas de una variable de entrada y con las de una variable de salida. Un ejemplo de una variable IN OUT es un cursor variable.

Parámetros IN

En el ejemplo anterior todas las variables p_car y p_cuantos son de tipo IN; como es la opción por defecto, no es necesario especificarlo.

Vamos a ver el uso de valores por defecto para parámetros IN, a través de un ejemplo.

Al procedimiento IMP_LIN le vamos a hacer una mejora, si no nos pasan los caracteres, le ponemos por defecto el carácter '-' (guion normal), y si no me pasan cuantos caracteres pongo por defecto 40.

```

CREATE OR REPLACE PROCEDURE IMP_LIN(p_car in char default '-', p_cuantos IN
pls_integer default 40) AS
BEGIN
    dbms_output.put_line(RPAD(p_car, p_cuantos, p_car));
END IMP_LIN;

```

Probamos el procedimiento con un bloque anónimo. !!!!OJO!!!! : si paso un parámetro sólo, PL toma el literal como si fuera un(os) carácter(es), el decir el numero 20, lo toma como '20', y te imprime 40 caracteres(2020202020...).

Si quiero pasar sólo el segundo parámetro, y que el carácter lo tome por defecto, fíjate en la llamada: `imp_lin(p_cuantos=> 10);`

Si invoco sin parámetros, toma los dos por defecto.

```

set serveroutput on

begin

    imp_lin('=',60);
    imp_lin('--FIN',30);
    imp_lin(20);
    imp_lin(p_cuantos=> 10);
    imp_lin('hola-');
    imp_lin;

end;
/
=====
--FIN--FIN--FIN--FIN--FIN--FIN
202020202020202020202020202020
-----
hola-hola-hola-hola-hola-hola-hola-
-----
```

Procedimiento PL/SQL terminado correctamente.

Parámetros OUT

Vamos a ver cómo se trabaja a través de un ejemplo.

Procedimiento llamado PAR_IMPARI, nos pasan por IN un número, y una variable VARCHAR2 en donde le decimos si es PAR o IMPAR. pero si el número es mayor de 100, levantamos una excepción de tipo RAISE_APPLICATION_ERROR(-20100,'numero excede de 100'). Por motivos didácticos primero asignamos el literal y luego levantamos la excepción.

Primero especificamos la variable como OUT, y vemos el efecto en el bloque anónimo.

```
create or replace PROCEDURE PAR_IMPARI(P_NUMERO PLS_INTEGER,
                                         P_LITERAL OUT VARCHAR2) AS

BEGIN
    IF MOD(P_NUMERO,2) = 0 THEN
        P_LITERAL := ' PAR';
    ELSE
        P_LITERAL := ' IMPAR';
    END IF;

    IF P_NUMERO > 100 THEN
        RAISE_APPLICATION_ERROR(-20100, 'EL NUMERO ES MAYOR DE 100');
    END IF;

END PAR_IMPARI;
```

Procedimiento PAR_IMPARI

Para probar el Procedimiento PAR_IMPARI, montamos un bucle desde el numero 98 al 101.

- En cada interacción ponemos V_LITERAL := NULL, a nulo, para ver el efecto.
- Los números 98,99,100, se ejecutan bien.

- Al llegar a 101, el procedimiento levanta la excepción -20100, el bloque anónimo la captura en WHEN OTHERS THEN, saca el mensaje y al escribir V_LITERAL, está a nulo, porque el procedimiento no ha volcado el literal 'IMPAR' almacenado en su variable P_LITERAL.
- La función escalar NVL aplicado a una variable, si esta tiene el nulo activado, muestra el literal asociado, si tiene contenido, muestra el contenido.

```

SET SERVEROUTPUT ON
DECLARE
    V_LITERAL VARCHAR2(40);

BEGIN
    FOR NUMERO IN 98 .. 101 LOOP
        PAR_IMPAR(NUMERO, V_LITERAL);
        DBMS_OUTPUT.PUT_LINE('EL NUMERO ' || NUMERO || ' ES ' || v_LITERAL);
        V_LITERAL := NULL;
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERROR EN BLOQUE PPAL : ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('LITERAL : ' ||
            nvl(v_LITERAL, 'ESTA A NULO EL P_LITERAL NO SE HA COPIADO'));
END;
/
EL NUMERO 98 ES PAR
EL NUMERO 99 ES IMPAR
EL NUMERO 100 ES PAR
ERROR EN BLOQUE PPAL : ORA-20100: EL NUMERO ES MAYOR DE 100
LITERAL : ESTA A NULO EL P_LITERAL NO SE HA COPIADO

```

Procedimiento PL/SQL terminado correctamente.

Bloque anónimo probar PAR_IMPAR

OUT NO COPY

Modificamos el Procedimiento PAR_IMPAR, e incrustamos NOCOPY detrás de OUT en P_LITERAL.

```

create or replace PROCEDURE PAR_IMPAR(P_NUMERO PLS_INTEGER,
                                      P_LITERAL OUT NOCOPY VARCHAR2) AS

BEGIN
  IF MOD(P_NUMERO, 2) = 0 THEN
    P_LITERAL := ' PAR';
  ELSE
    P_LITERAL := ' IMPAR';
  END IF;

  IF P_NUMERO > 100 THEN
    RAISE_APPLICATION_ERROR(-20100, 'EL NUMERO ES MAYOR DE 100');
  END IF;

END PAR_IMPAR;

```

Y ejecutamos el mismo bloque anónimo anterior (Bloque anónimo probar PAR_IMPAR).

El resultado:

```

Procedimiento PL/SQL terminado correctamente.
EL NUMERO 98 ES PAR
EL NUMERO 99 ES IMPAR
EL NUMERO 100 ES PAR
ERROR EN BLOQUE PPAL : ORA-20100: EL NUMERO ES MAYOR DE 100
LITERAL : IMPAR

```

Procedimiento PL/SQL terminado correctamente.

Date cuenta en la última línea, ha salido impar (era el numero 101).

Es decir con la opción OUT NOCOPY, el procedimiento trabaja directamente con la variable que le paso: V_LITERAL, la modifica y pase lo que pase en el procedimiento, cuando nos cede el control, mi variable tiene el contenido (en este caso 'IMPAR').

- i** En el bloque anónimo que hemos empleado para este ejemplo (Bloque anónimo probar PAR_IMPAR) de variables OUT, ni se te ocurra en el DBMS de la excepción referenciar la variable "numero" del FOR, porque recuerda que esa variable es "local" a FOR, y te daría error de interpretación.

```
.....
FOR NUMERO IN 98 .. 101 LOOP
.....
END LOOP;

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERROR EN BLOQUE PPAL : ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('LITERAL : ' || nvl(v_LITERAL,'ESTA A NULO EL
P_LITERAL NO SE HA COPIADO'));
        DBMS_OUTPUT.PUT_LINE('EL NUMERO ' || NUMERO || ' ES ' || v_LITE-
RAL);

    END;
/
Informe de error -
ORA-06550: line 14, column 46:
PLS-00201: identifier 'NUMERO' must be declared
ORA-06550: line 14, column 9:
PL/SQL: Statement ignored
06550. 00000 - "line %s, column %s:\n%s"
*Cause: Usually a PL/SQL compilation error.
*Action:
```

Cláusula PRAGMA AUTONOMOUS_TRANSACTION

Un procedimiento autónomo permite realizar COMMIT o ROLLBACK de las sentencias SQL propias sin afectar a la transacción que lo haya llamado. Si en ejecución no se encuentra la sentencia COMMIT o ROLBAK provocará una excepción.

En el ejemplo siguiente, el COMMIT sólo afecta al INSERT del procedimiento llamado.

```

CREATE OR REPLACE
PROCEDURE Llamador
IS
BEGIN
    UPDATE departments
    SET location_id=1700;           -- Se inicia una transacción
    --Se invoca un procedimiento autónomo
    Procedimiento_autonomo;

    ROLLBACK;                     -- Se deshará el UPDATE no el INSERT.
END Llamador;
/

CREATE OR REPLACE
PROCEDURE Procedimiento_autonomo
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO locations(location_id,city)
    VALUES (locations_seq.nextval,'Madrid');

    COMMIT; /* Sólo afectará al INSERT del procedimiento autónomo*/
END Procedimiento_autonomo;
/

```

Funciones Almacenadas

Definición formal

Una función es un subprograma que calcula un valor. Las funciones difieren principalmente de los procedimientos en que siempre retornan un valor mediante la instrucción RETURN.

La sintaxis para crear una función es:

```
CREATE [OR REPLACE]
FUNCTION Nombre_Función
    [(declaracion_parámetro
      [, declaracion_parámetro]...)]
RETURN Tipo_dato
    [AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
    [DETERMINISTIC]
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [Declaraciones locales de tipos, variables, etc]
BEGIN
    /*Sentencias ejecutables*/
    .....
    return literal/variable;
[EXCEPTION
    --Excepciones definidas y las acciones de estas excepciones]
    .....
    return literal/variable;
END [Nombre_Función];
```

Su estructura es exactamente igual a los procedimientos, con estas dos diferencias:

1

En la cabecera de la función y antes del IS/AS, incorporan la cláusula RETURN tipo de dato. El tipo de datos es : tipos de Oracle, más pls_inteber, más boolean, mas variables %Type, más registros%Rowtype.

2

La cláusula DETERMINISTIC: ayuda al optimizador de Oracle a evitar llamadas redundantes. Si una función almacenada ha sido anteriormente invocada con los mismos parámetros el optimizador puede escoger devolver el mismo valor, sin volver a ejecutar la función.

3

Toda Función finaliza con una sentencia RETURN, en donde se devuelve el contenido de una variable y/o un literal correspondiente.

La Función más pequeña que podemos hacer sin errores de compilación es:

```
CREATE OR REPLACE FUNCTION NOMBRE_FUNCION RETURN VARCHAR2 AS
BEGIN
    RETURN NULL;
END NOMBRE_FUNCION;
```

Cuerpo vacío de Función

Definición y ejecución de funciones

Lo vemos con un ejemplo.

Vamos a hacer una función denominada EXISTE_DEP, que recibe como parámetro de entrada un código de departamento, y nos informa si ese departamento existe o no. Además, si el departamento existe, en un parámetro de tipo OUT nos vuelca un registro con todos los datos de este departamento.

Intenta hacerlo tú, mira la solución para comprobar el resultado.

Solución:

```

CREATE OR REPLACE FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE,
                                     F_DEP OUT DEPARTMENTS%ROWTYPE) RETURN BOOLEAN
AS
BEGIN
    SELECT *
    INTO F_DEP
    FROM DEPARTMENTS
    WHERE DEPARTMENT_ID = P_DEP;
    RETURN TRUE;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE;
END EXISTE_DEP;

```

Función EXISTE_DEP

Y ahora vamos a hacer un bloque anónimo para probar. Haremos tres invocaciones:

- departamento 30 : existe, y mostramos el nombre del departamento
- departamento 430 : mostramos el mensaje no existe

```

set serveroutput on
declare
    f_dep departments%rowtype;
begin
-- el primer parametro de la función lo cogemos de la consola
    if existe_dep(&dep, f_dep) then
        dbms_output.put_line('departamento : ' || f_dep.department_name);
    else
        dbms_output.put_line('departamento no existe');
    end if;
exception
    when others then
        dbms_output.put_line('error en bloque anónimo : ' || sqlerrm);
end;
/
con 30
departamento : Purchasing

Procedimiento PL/SQL terminado correctamente.

con 430

departamento no existe

Procedimiento PL/SQL terminado correctamente.

```

La sentencia RETURN (No la de la parte de la especificación de la función, donde especificamos el tipo de dato que se devuelve) finaliza la ejecución de la función y devuelve el control.

Una función puede contener varias sentencias RETURN.

La sentencia RETURN no tiene porque ser la última sentencia del subprograma o función.

En las funciones, la sentencia debe devolver un valor. Este valor se evalúa en el momento de devolverlo, por lo que puede ser una expresión.

Las funciones son invocadas como parte de una expresión y pueden ser invocadas desde múltiples sitios:

- Un IF, para evaluar la condición.
- Un bucle, para evaluar su finalización.
- Para asignar lo que devuelve la función a una variable.

