INNOV@
Desarrollos Informáticos

PL/SQL
Básico

Manual del Alumno

INNOV@
Desarrollos Informáticos

INNOVA Desarrollos Informáticos, SL
Paseo Mallorca, 34 Entlo. C
07012 Palma de Mallorca
Tel. 971 72 14 04

Título:	PL/SQL Básico
Versión:	1.0
Fecha Edición:	Junio de 2004
Autores:	Javier Jofre González-Granda

INTRODUCCIÓN

Prólogo

PL/SQL, bajo este nombre se esconde el Lenguaje de manipulación de datos propietario de Oracle. Conceptualmente, Oracle lo define como una extensión procedimental del SQL… en realidad, y para entenderlo mejor, se trata de un potente lenguaje de acceso a Bbdd, mediante el cual podemos estructurar y controlar las sentencias SQL que definamos para nuestra Bbdd.

PL/SQL sigue la filosofía de los modernos lenguajes de programación, es decir, permite definir y manipular distintos tipos de datos, crear procedimientos, funciones, contempla recursividad, etc… Quizás la diferencia más importante, y algo que debemos tener siempre muy en cuenta, es que la eficiencia de un programa en PL/SQL se mide sobre todo por la eficiencia de los accesos a Bbdd.

La consecuencia más inmediata de lo dicho anteriormente, es que para poder programar de manera óptima en PL/SQL, se debe tener un dominio notable del propio SQL; cumpliendo esta premisa, y algunas otras que veremos más adelante, obtendremos una mejora sustancial en nuestras aplicaciones que interactuen con Bbdd.

ÍNDICE

INNOV@
Desarrollos Informáticos

FICHA INFORMATIVA DEL MÓDULO.

Nombre

PL/SQL Básico

Meta

Que el Alumno adquiera los conocimientos básicos sobre estructuras de datos y sentencias, necesarios para el desarrollo de aplicaciones que llamen a subprogramas PL/SQL en el acceso a Bbdd.

Requisitos del alumno

Poseer conocimientos de Bbdd, así como del lenguaje SQL utilizado por la plataforma Oracle. También es necesario conocer mínimamente los fundamentos de la Programación estructurada.

Bibliografía

PL/SQL User's Guide and Reference, y varios artículos sobre PL/SQL obtenidos de Internet.

INNOV@
Desarrollos Informáticos

1 UNIDAD 4:INTRODUCCIÓN A PL/SQL

Objetivo general de la unidad

Asimilar los conceptos básicos que se manejan dentro de la programación en PL/SQL.

Objetivos específicos

Conocer los tipos de datos soportados por PL/SQL, así como la sintaxis básica de las sentencias que utiliza.

Contenidos

INNOV@
Desarrollos Informáticos

Cuaderno de notas

1.1 Introducción

Cuando escribimos un programa en PL/SQL, utilizamos un conjunto específico de caracteres. El conjunto de caracteres soportado es el siguiente:

- Los caracteres mayúsculas y minúsculas A … Z, a … z
- Los números 0 … 9
- Tabulaciones, espacios y retornos de carro
- Los símbolos ( ) + - * / < > = ¡ ~ ; : . ‘ @ % , “ # $ ^ & _ | { } ¿ [ ]

PL/SQL no es ‘case sensitive’, por lo tanto no distingue entre mayúsculas y minúsculas, excepto para un par de casos que comentaremos más adelante.

1.1.1 Unidades Léxicas

Una sentencia de PL/SQL contiene grupos de caracteres, llamados *Unidades Léxicas*, las cuales se clasifican de la siguiente forma:

- Delimitadores (Símbolos simples y compuestos)
- Identificadores, los cuales incluyen a las palabras reservadas
- Literales
- Comentarios

Por ejemplo, la siguiente sentencia:

*bonificacion := salario * 0.10; -- Cálculo de Bonus*

contiene las siguientes unidades léxicas:

- identificadores: bonificacion y salario
- símbolo compuesto: :=
- símbolos simples: * y ;
- literal numérico: 0.10
- comentario: -- Cálculo de Bonus

Para mejorar la lectura de un código fuente, podemos (y de hecho debemos) separar las unidades léxicas por Espacios o Retornos de Carro, siempre manteniendo las reglas básicas del lenguaje.

Por ejemplo, la siguiente sentencia es válida:

IF x>y THEN max:=x;ELSE max:=y;END IF;

Sin embargo, deberíamos escribirla así para facilitar su lectura:

```
IF x>y THEN
    max:=x;
ELSE
    max:=y;
END IF;
```

Vamos a ver en detalle cada una de las Unidades Léxicas.

1.1.2 Delimitadores

Un *delimitador* es un símbolo simple o compuesto, que tiene un significado especial en PL/SQL.

Veamos cada uno de los tipos.

1.1.2.1 Símbolos simples

La lista y significado de los símbolos simples son los siguientes:

+	Operador de suma
%	Indicador de Atributo
‘	Carácter delimitador de String
.	Selector
/	Operador de división
(	Expresión o delimitador de lista
)	Expresión o delimitador de lista
:	Indicador de variable host
,	Separador de Items
*	Operador de multiplicación
“	Delimitador de identificadores
=	Operador relacional
<	Operador relacional
>	Operador relacional
@	Indicador de acceso remoto
;	Terminador de sentencia
-	Resta/Operador de negación

1.1.2.2 Símbolos compuestos

La lista y significado de símbolos compuestos son los siguientes:

** Operador de exponenciación

<> Operador relacional

!= Operador relacional

~= Operador relacional

<= Operador relacional

>= Operador relacional

:= Operador de Asignación

=> Operador de asociación

.. Operador de rango

|| Operador de concatenación

<< (Comienzo) delimitador de etiqueta

>> (Fin) delimitador de etiqueta

-- Indicador de comentario para una sola línea

/* (Comienzo) delimitador de comentario de varias líneas

*/ (Fin) delimitador de comentario de varias líneas

1.1.3 Identificadores

Los identificadores se utilizan para dar nomenclatura a unidades e items de un programa PL/SQL, el cual puede incluir constantes, variables, excepciones, cursores, cursores con variables, subprogramas y packages.

Un identificador consiste en una letra seguida, de manera opcional, de más letras, números, signos de dólar, underscores, y signos numéricos. Algunos caracteres como % - / y espacios son ilegales.

Ejemplo:

```
mi_variable     -- Identificador legal
mi variable     -- Identificador Ilegal
mi-variable     -- Identificador Ilegal
```

Se pueden usar mayúsculas, minúsculas, o mezcla de ambas… ya hemos comentado que PL/SQL no es 'case sensitive', con lo cual no las diferenciará, exceptuando el caso en que estemos ante tratamiento de Strings, o bien literales de un solo carácter.

Veamos algún ejemplo:

```
minombre
MiNombre          -- Igual que minombre
MINOMBRE          -- Igual que minombre
```

La longitud de un identificador no puede exceder los 30 caracteres.

Por supuesto, y esto casi obvia decirlo, puesto que sigue las reglas básicas de la programación, los identificadores **deben ser siempre descriptivos**.

1.1.3.1 Palabras Reservadas

Algunos identificadores, llamados *Palabras Reservadas*, tienen un significado sintáctico especial para PL/SQL, y no pueden ser redefinidas; un claro ejemplo son las palabras BEGIN y END.

```
DECLARE
 end BOOLEAN;          -- Ilegal

DECLARE
 end_film BOOLEAN;     -- Legal
```

Las palabras reservadas se suelen poner en mayúsculas, para facilitar la lectura del código fuente.

1.1.3.2 Identificadores Predefinidos

Los identificadores globales declarados en el package STANDAR, como por ejemplo la excepción INVALID_NUMBER, pueden ser redeclarados… sin embargo, la declaración de identificadores predefinidos es un error, puesto que las declaraciones locales prevalecen sobre las globales.

1.1.3.3 Identificadores con Comillas Dobles

Por Flexibilidad, PL/SQL permite incluir identificadores con dobles comillas. Estos identificadores no son necesarios muy a menudo, pero a veces pueden ser de gran ayuda.

Pueden contener cualquier secuencia de caracteres, incluyendo espacios, pero excluyendo las comillas dobles. Veamos algunos ejemplos:

"X+Y"

"ultimo nombre"

"switch on/off"

La longitud máxima para este tipo de identificadores es de 30 caracteres. Aunque se permite, la utilización de palabras reservadas por PL/SQL como identificadores con doble comillas, es una mala práctica.

Hemos dicho, no obstante, que en algunas ocasiones nos puede venir muy bien su uso… veamos un ejemplo:

Algunas palabras reservadas en PL/SQL no son palabras reservadas en SQL. Por ejemplo, podemos usar la palabra reservada en PL/SQL TYPE en un CREATE TABLE para llamar así a una columna de la tabla. Pero si definimos una sentencia de acceso a dicha tabla en PL/SQL de la siguiente forma:

SELECT nom,type,bal INTO …

Nos provocará un error de compilación… para evitar esto podemos definir la sentencia de la siguiente manera:

SELECT nom,"TYPE",bal INTO …

Así nos funcionará… es importante hacer notar que siempre, en un caso como el del ejemplo, deberemos poner el identificador en mayúsculas.

1.1.4 Literales

Un *literal* es un valor explícito de tipo numérico, carácter, string o booleano, no representado por un identificador. El literal numérico 147, y el literal booleano FALSE, son ejemplos de esto.

1.1.4.1 Literales Numéricos

Podemos utilizar dos clases de literales numéricos en expresiones aritméticas: enteros y reales.

Un literal Entero, es un número Entero, al que podemos opcionalmente poner signo. Ejemplos:

30, 6, -14, 0, +32000, …

Un literal Real, es un número Entero o fraccional, con un punto decimal. Ejemplos son:

6.667, 0.0, -12.0, +86.55, …

PL/SQL considera como reales a los números de este tipo 12.0, 25. , aunque tengan valores enteros. Es muy importante tener esto en cuenta, básicamente porque **debemos evitar al máximo trabajar con reales si podemos hacerlo con enteros, ya que eso disminuye la eficiencia de nuestro programa**.

Los literales numéricos no pueden tener los signos dólar o coma, sin embargo, pueden ser escritos en notación científica. Ejemplo:

2E5, 1.0E-7, …

1.1.4.2 Literales de tipo Carácter

Un literal de tipo Carácter, es un carácter individual entre comillas simples. Por ejemplo:

'Z', '%', '7', …

Los literales de tipo Carácter, incluyen todo el conjunto de caracteres válidos en PL/SQL.

PL/SQL es 'case sensitive' con los literales de tipo carácter. Por ejemplo 'Z' y 'z' son diferentes.

Los literales de tipo carácter '0' … '9', no son equivalentes a literales numéricos.. sin embargo, pueden ser utilizados en expresiones aritméticas gracias a la conversión de tipos implícita de PL/SQL.

1.1.4.3 Literales de tipo String

Un valor de tipo carácter puede ser representado por un identificador, o de forma explícita escrito como un literal de tipo String, el cual es una secuencia de cero o más caracteres delimitados por comillas simples. Ejemplos:

'Mi nombre es Pepe', '10-ENE-2000', '$100000', …

Si deseamos que la cadena de caracteres tenga una comilla simple, lo que debemos hacer es repetir la comilla simple. Ejemplo:

'Don''t leave without saving your work'

PL/SQL es 'case sensitive' para los literales de tipo String.

1.1.4.4 Literales de tipo Booleano

Los literales de tipo Booleano son los valores predeterminados TRUE, FALSE, y NULL en el caso de no tener ningún valor. Recordemos siempre que los literales de tipo Booleano son valores y no Strings.

1.1.5 Comentarios

El compilador de PL/SQL ignora los comentarios, sin embargo nosotros no debemos hacerlo. La inserción de comentarios es muy útil a la hora de entender un programa.

PL/SQL soporta dos tipos de comentarios: los de una línea, y los de múltiples líneas.

1.1.5.1 Comentarios de una línea

Se definen mediante dos signos menos consecutivos (--), que se ponen al principio de la línea a comentar… a partir de ahí, y hasta la línea siguiente, se ignora todo. Ejemplo:

```
-- Comienzo mi Select
SELECT * FROM mitabla
 WHERE mit_mitkey=p_codigo ;  -- Solo los de mi código
```

1.1.5.2 Comentarios de múltiples líneas

Se define el comienzo mediante un slash y un asterisco (/*), y el fin mediante un asterisco y un slash (*/); todo lo incluido entre comienzo y fin comentario, será ignorado. Ejemplo:

```
BEGIN
 SELECT COUNT(*) INTO contador FROM PERSONAS;
 /* Si el resultado es mayor que cero, actualizaremos
  el histórico de personas */
 IF contador>0 THEN ….
```

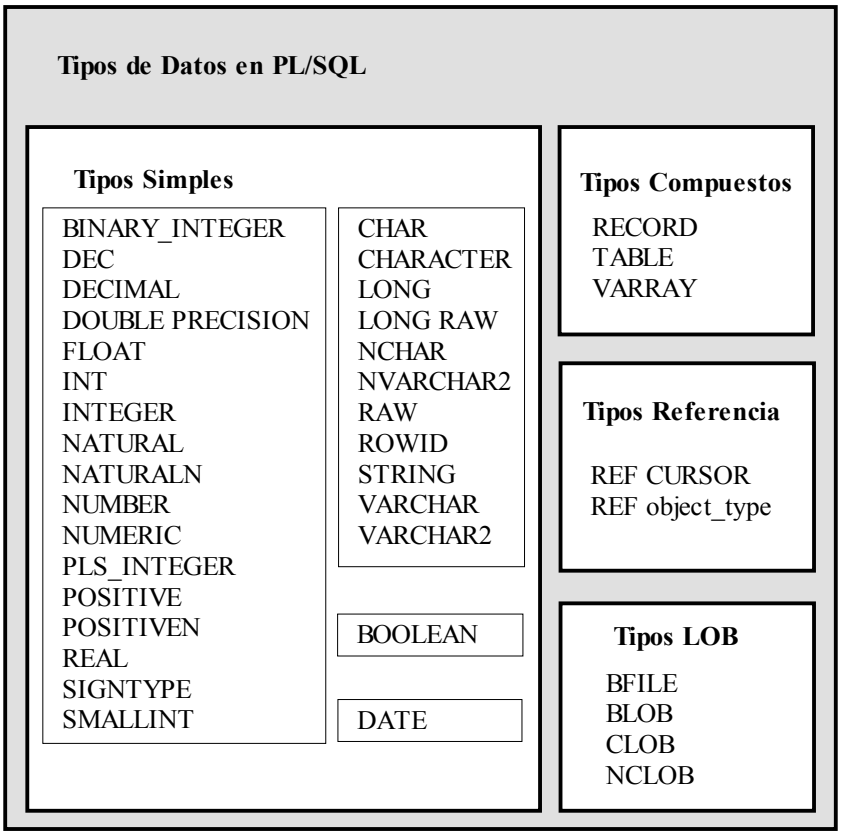
1.2 Tipos de Datos

En este apartado, nos limitaremos a dar una tabla con todos los tipos existentes en PL/SQL, y explicaremos algunos que sean de interés.

La razón es que los tipos más utilizados coinciden al 100% con los del SQL de Oracle, y por tanto son conocidos por los asistentes a este curso.

Además, aquellos tipos que sean específicos para alguna funcionalidad concreta que soporte PL/SQL, serán vistos en detalle cuando abordemos cada una de esas funcionalidades.

Los tipos soportados por PL/SQL son los siguientes:



Como notas de Interés, decir que los tipos NATURAL y POSITIVE permiten la restricción a valores que sean tan solo positivos, mientras que NATURALN y POSITIVEN, evitan la asignación de valores NULL.

El tipo SIGNTYPE permite la restricción de una variable a los valores –1, 0, y 1, lo cual es útil a la hora de programar lógica.

NUMBER es un tipo que tiene el siguiente rango:

1.0E-130 … 9.99E125

INTEGER tiene una precisión de 38 dígitos decimales.

PLS_INTEGER es un tipo especial de PL/SQL, equivalente a INTEGER, pero que nos da una eficiencia mucho mayor, por tanto es importante **utilizarlo siempre que tratemos enteros en nuestros programas**.

El tipo LONG, aunque pretende ser equivalente al tipo LONG de SQL, en realidad solo admite hasta 32.760 caracteres, si queremos trabajar con columnas de tipo LONG, deberemos utilizar otro tipo de técnicas como SQL Dinámico.

El tipo VARCHAR2 también admite hasta 32.760 caracteres.

1.2.1 Subtipos definidos por el usuario

Cada tipo básico de PL/SQL, especifica un conjunto de valores, así como un conjunto de operaciones aplicables sobre los mismos. Los subtipos, nos permitirán especificar las mismas operaciones que las aplicables a los tipos básicos, **pero tan solo sobre un subconjunto de valores**.

1.2.2 Definición de Subtipos

Podemos definir nuestros propios Subtipos en la parte de declaraciones de cualquier bloque PL/SQL, subprograma o package, utilizando la sintaxis:

SUBTYPE nombre_subtipo IS tipo_base;

Donde nombre_subtipo es el nombre que se desee darle, y tipo_base es cualquier tipo de PL/SQL predefinido o definido por el usuario. Para especificar el tipo_base, podemos usar %TYPE, el cual proporciona el tipo de datos de una variable, o una columna de Bbdd, o también %ROWTYPE, que nos proporciona el tipo ROW de un cursor, cursor de variables, o tabla de Bbdd. Veamos algunos ejemplos de definición:

```
DECLARE
  SUBTYPE FechaEmp IS DATE;          -- Basado en un Tipo DATE
  SUBTYPE Contador IS NATURAL;       -- Basado en un subtipo NATURAL
  TYPE ListaNombres IS TABLE OF VARCHAR2(10);
  SUBTYPE NomEmp IS ListaNombres;    -- Basado en un tipo TABLE
  TYPE TimeRec IS RECORD(minutos INTEGER,horas INTEGER);
  SUBTYPE Time IS TimeRec;           -- Basado en un tipo RECORD
  SUBTYPE Id_Num IS emp.numemp%TYPE; -- Basado en un tipo columna
  CURSOR c1 IS SELECT * FROM dep;
  SUBTYPE Depsub IS c1%ROWTYPE;      -- Basado en una Row de un Cursor
```

Sin embargo, no podemos especificar constraints sobre el tipo base. Veamos algunas declaraciones ilegales:

```
DECLARE
  SUBTYPE Acumulacion IS NUMBER(7,2);   -- Ilegal
  SUBTYPE Palabra IS VARCHAR2(15);      -- Ilegal
```

Aunque no podamos especificar constraints de forma directa, en realidad podemos hacerlo de manera indirecta de la siguiente manera:

```
DECLARE
  temp VARCHAR2(15);
  SUBTYPE Palabra IS temp%TYPE;   -- La longitud máxima de Palabra será 15
```

También debe mencionarse, que si se define un subtipo utilizando %TYPE para proporcionar el tipo de dato de una columna de Bbdd, el subtipo adopta la constraint de longitud de la columna, sin embargo, el subtipo no adoptará otro tipo de constraints como NOT NULL.

1.2.3 Utilizando Subtipos

Una vez que se ha declarado un subtipo, podemos declarar items de ese tipo. Veamos un par de ejemplos:

```
DECLARE
  SUBTYPE Contador IS NATURAL;
  rows      Contador;
  empleados      Contador;
  SUBTYPE Acumulador IS NUMBER;
  total Acumulador(7,2);
```

Los subtipos pueden ayudar en determinados casos al tratamiento de errores, si se definen adecuadamente dentro de algún rango. Por ejemplo si tenemos una variable y sabemos que su rango será –9 … 9, podemos hacer la definición de la siguiente forma.

```
DECLARE
  temp NUMBER(1,0);
  SUBTYPE Escala IS temp%TYPE;
  eje_x Escala;     -- El rango será entre –9 y 9
  eje_y Escala;
BEGIN
  eje_x := 10;              -- Esto nos provocará un VALUE_ERROR
```

1.2.3.1 Compatibilidad de Tipos

Un Subtipo siempre es compatible con su tipo base. Por ejemplo, en las siguientes líneas de código, no es necesaria ninguna conversión:

```
DECLARE
  SUBTYPE Acumulador IS NUMBER;
  cantidad NUMBER(7,2);
  total Acumulador;
BEGIN
  …
  total := cantidad;
```

También son compatibles diferentes subtipos, siempre y cuando tengan el mismo tipo base. Ejemplo:

```
DECLARE
  SUBTYPE Verdad IS BOOLEAN;
  SUBTYPE Cierto IS BOOLEAN;
  miverdad Verdad;
  micierto Cierto;
BEGIN
  …
  micierto := miverdad;
```

Por último, subtipos diferentes también son compatibles en el supuesto de que sus tipos base sean de la misma familia de tipos de dato. Ejemplo:

```
DECLARE
  SUBTYPE Palabra IS CHAR;
  SUBTYPE Texto IS VARCHAR2;
  verbo Palabra;
  sentencia Texto;
BEGIN
  …
  sentencia := verbo;
```

1.2.4 Conversiones de Tipos

A veces es necesario realizar la conversión de un valor, de un tipo de dato a otro. Por ejemplo, si se desea comprobar el valor de un ROWID, debemos convertirlo a un string de caracteres. PL/SQL soporta tanto la conversión explícita de datos, como la implícita (automática).

1.2.5 Conversión Explícita

Para convertir valores de un tipo de datos a otro, se debe usar funciones predefinidas. Por ejemplo, para convertir un CHAR a un tipo DATE o NUMBER, debemos utilizar las funciones TO_DATE o TO_NUMBER, respectivamente. De forma análoga, para convertir un tipo DATE o NUMBER a CHAR, debemos utilizar la función TO_CHAR.

1.2.6 Conversión Implícita

Cuando tiene sentido, PL/SQL puede convertir de forma implícita un tipo de dato a otro. Esto nos permite utilizar literales, variables y parámetros de un tipo, en lugares donde se espera otro tipo. Veamos un ejemplo:

```
DECLARE
  tiempo_comienzo CHAR(5);
  tiempo_fin CHAR(5);
  tiempo_transcurrido NUMBER(5);
BEGIN
  /* Obtenemos la hora del sistema como segundos */
  SELECT TO_CHAR(SYSDATE,'SSSSS') INTO tiempo_comienzo
    FROM sys.dual;
  /* Volvemos a obtenerla */
  SELECT TO_CHAR(SYSDATE,'SSSSS') INTO tiempo_fin
    FROM sys.dual;
  /* Calculamos el tiempo transcurrido en segundos */
  tiempo_transcurrido := tiempo_fin – tiempo_comienzo;
  …
END;
```

Antes de asignar el valor de una columna seleccionada a una variable, PL/SQL convertirá, si es necesario, el tipo de dato de la variable al tipo

de dato de la columna. Esto ocurre, por ejemplo, cuando se selecciona una columna tipo DATE en una variable tipo VARCHAR2.

En cualquier caso, y aunque PL/SQL lo permita y haga, debemos tener cuidado y evitarlas al máximo, puesto que la conversión implícita ralentizará nuestro programa. Las conversiones explícitas son mejores, y además nos aseguran un mantenimiento más sencillo del programa.

Veamos una tabla con todas las conversiones posibles:

	Bin_Int	Char	Date	Long	Number	Pls_Int	Raw	Rowid	Varchar2
Bin_Int		X		X	X	X			X
Char	X		X	X	X	X	X	X	X
Date		X		X					X
Long		X					X		X
Number	X	X		X		X			X
Pls_Int	X	X		X	X				X
Raw		X		X					X
Rowid		X							X
Varchar2	X	X	X	X	X	X	X	X	

Es responsabilidad del programador, asegurarse que los valores son convertibles, por ejemplo, PL/SQL puede convertir el valor CHAR '02-JUN-92' a un tipo DATE, pero no puede convertir el valor CHAR 'YESTERDAY' a un valor DATE. De forma similar, PL/SQL no puede convertir un valor VARCHAR2 que contenga caracteres alfabéticos a un valor de tipo NUMBER.

1.3 Declaraciones

En PL/SQL se pueden declarar tanto constantes como variables; recordemos que **las variables pueden cambiar en tiempo de ejecución, mientras que las constantes permanecen con el mismo valor de forma continua**.

Se pueden declarar constantes y variables en la parte de declaración de cualquier bloque PL/SQL, subprograma, o package. Las declaraciones reservan espacio para un valor en función de su tipo.

Al hacer la declaración, daremos un nombre a la variable o constante, para de esta forma poder referenciarla a lo largo de la ejecución del Programa.

Veamos un par de ejemplos de declaraciones:

Cumple DATE;

Cuenta SMALLINT := 0;

Como vemos, al declarar una variable o constante, podemos darle un valor inicial. Incluso podemos asignarle expresiones, como en el siguiente ejemplo:

pi REAL := 3.14159;

radio REAL := 1;

*area REAL := pi*radio*2;*

Por defecto, las variables se inicializan a NULL, así que las siguientes dos declaraciones serían equivalentes:

cumple DATE;

cumple DATE := NULL;

Cuando declaremos una constante, la palabra clave CONSTANT debe preceder a la especificación del tipo. Veamos un ejemplo:

limite_de_credito CONSTANT REAL := 250.000;

1.3.1 Utilizando DEFAULT

Se puede utilizar la palabra clave DEFAULT, en lugar del operador de asignación, para inicializar variables. Por ejemplo, las siguientes declaraciones:

tipo_sangre CHAR := 'O';

valido BOOLEAN := FALSE;

Pueden ser escritas de la siguiente manera:

tipo_sangre CHAR DEFAULT 'O';

valido BOOLEAN DEFAULT FALSE;

Se utiliza DEFAULT para las variables que tienen un valor típico, mientras que el operador de asignación, se usa en aquellos casos en que las variables no tienen dicho valor, como por ejemplo en contadores y acumuladores. Veamos un ejemplo:

horas_trabajo INTEGER DEFAULT 40;

contador INTEGER:=0;

1.3.2 Utilizando NOT NULL

Además de asignar un valor inicial, en una declaración se puede imponer la constraint de NOT NULL. Veamos un ejemplo:

```
id_acc INTEGER(4) NOT NULL := 9999;
```

Evidentemente, no se pueden asignar valores NULL a variables que se han definido como NOT NULL, de hecho, si intentamos hacerlo, PL/SQL dará la excepción predefinida VALUE_ERROR.

La constraint NOT NULL, debe estar seguida por una claúsula de inicialización. Por ejemplo, la siguiente declaración no es válida:

```
id_acc INTEGER(4) NOT NULL;  -- Falta la inicialización…
```

Recordemos que los subtipos NATURALN y POSITIVEN, ya están predefinidos como NOT NULL.

```
cont_emp NATURAL NOT NULL := 0;

cont_emp NATURALN := 0;  -- La sentencia de arriba y esta, son equivalentes…

cont_emp NATURALN;  -- Declaración Ilegal, falta la inicialización…
```

1.3.3 Utilizando %TYPE

El atributo %TYPE, proporciona el tipo de dato de una variable o de una columna de la Bbdd. En el siguiente ejemplo, %TYPE asigna el tipo de dato de una variable:

```
credito REAL(7,2);

debito credito%TYPE;
```

La declaración utilizando %TYPE, puede incluir una claúsula de inicialización. Veamos un ejemplo:

```
balance NUMBER(7,2);

balance_minimo balance%TYPE := 10.00;
```

De todas formas, el uso de %TYPE es especialmente útil en el caso de definir variables que sean del tipo de una columna de la Bbdd. Veamos un ejemplo:

```
el_nombre globalweb.usuarios.usu_nomusu%TYPE;
```

Fijémonos en que la utilización de este tipo de declaración tiene dos claras ventajas: por un lado **no es necesario conocer el tipo de dato que tiene la columna de la tabla**, y por otro, **si cambiamos el tipo de dato de la columna, no deberemos modificar el PL/SQL**.

En cuanto a los inconvenientes, debe mencionarse el hecho de que la utilización de este tipo de declaraciones ralentiza un poco la ejecución del PL/SQL. Por tanto su uso debe estar siempre justificado, como por ejemplo en el caso de una columna que pueda ser susceptible de modificación.

Como último apunte, decir que el hecho de asignar a una variable el tipo de dato de una columna que tenga la constraint de NOT NULL utilizando %TYPE, NO nos aplicará dicha constraint a la variable. Veamos un ejemplo:

```
DECLARE
  num_emp emp.id_emp%TYPE;
  …
BEGIN
  num_emp := NULL;  -- No nos dará ningun error…
  …
END;
```

1.3.4 Utilizando %ROWTYPE

El atributo %ROWTYPE proporciona un tipo 'registro', que representa una fila de una tabla (o una vista). El registro puede almacenar toda la fila de una tabla (o de un cursor sobre esa tabla), o bien una serie de campos recuperados mediante un cursor. Veamos un par de ejemplos que ilustren esto:

```
DECLARE
  emp_rec emp%ROWTYPE;
  CURSOR c1 IS
    SELECT num_dept, nom_dept, dir_dept
      FROM dept;
  dept_rec c1%ROWTYPE;
```

Las columnas de una fila, y los correspondientes campos del registro, tienen los mismos nombres y tipos de dato. En el siguiente ejemplo, vamos a seleccionar los valores de una fila en un registro llamado *emp_rec*:

```
DECLARE
 emp_rec emp%ROWTYPE;
 …
BEGIN
 SELECT * INTO emp_rec
  FROM emp
   WHERE ROWNUM=1;
 …
END;
```

Para referenciar los valores almacenados en un registro, utilizaremos la notación siguiente:

nombre_registro.nombre_campo

Por ejemplo, en el caso anterior referenciaríamos al campo nombre de la siguiente manera:

IF emp_rec.emp_nomemp='Perico' THEN …

1.3.4.1 Asignaciones entre registros

Una declaración del tipo %ROWTYPE, no puede incluir una claúsula de inicialización, sin embargo, existen dos maneras de asignar valores a todos los campos de un registro a la vez.

En primer lugar, PL/SQL permite la asignación entre registros de forma completa, siempre y cuando su declaración referencie a la misma tabla o cursor. Veamos un ejemplo:

```
DECLARE
 dept_rec1 dept%ROWTYPE;
 dept_rec2 dept%ROWTYPE;
 CURSOR c1 IS
  SELECT num_dept, nom_dept, dir_dept
   FROM dept;
 dept_rec3 c1%ROWTYPE;
BEGIN
 …
 dept_rec1 := dept_rec2;
```

Esto que hemos hecho es válido, sin embargo y debido a que dept_rec2 se basa en una tabla, y dept_rec3 se basa en un cursor, la siguiente sentencia no sería válida:

```
dept_rec2 := dept_rec3;  -- Asignación no Válida…
```

Otra forma de realizar la asignación de una lista de valores de columnas a un registro, sería mediante la utilización de las sentencias SELECT o FETCH. Los nombres de las columnas deben aparecer en el orden en el que fueron definidas por las sentencias CREATE TABLE, o CREATE VIEW. Veamos un ejemplo:

```
DECLARE
 dept_rec dept%ROWTYPE;
 …
BEGIN
 SELECT num_dept, nom_dept, dir_dept INTO dept_rec
  FROM dept
   WHERE num_dept=30;
 …
END;
```

Sin embargo, no se puede asignar una lista de valores de columnas a un registro utilizando una sentencia de asignación. Por lo tanto, la siguiente sentencia no es válida:

```
nombre_registro := (valor1, valor2, valor3, …);  -- No Válido…
```

Por último, decir que aunque podemos recuperar registros de forma completa, no podemos realizar inserts o updates utilizando los mismos. Veamos un ejemplo:

```
INSERT INTO dept VALUES (dept_rec);  -- No Válido…
```

1.3.4.2 Utilizando Alias

Los elementos de una lista de tipo select, recuperada mediante un cursor que tiene asociado un %ROWTYPE, deben tener nombres simples o, **en el caso de ser expresiones, deben tener un alias**. Veamos un ejemplo en el cual utilizaremos un alias llamado *wages*:

```
DECLARE
 CURSOR mi_cursor IS
  SELECT salario + NVL(comm,0) wages, nom_emp
   FROM emp;
```

```
  mi_rec mi_cursor%ROWTYPE;
BEGIN
  OPEN mi_cursor;
  LOOP
    FETCH mi_cursor INTO mi_rec;
    EXIT WHEN mi_cursor%NOTFOUND;
    IF mi_rec.wages>2000 THEN
      INSERT INTO temp VALUES (NULL, mi_rec.wages,
mi_rec.nom_emp);
    END IF;
  END LOOP;
  CLOSE mi_cursor;
END;
```

1.3.5 Restricciones

PL/SQL **no permite referencias de tipo *forward***, es decir, se debe crear una variable o constante antes de referenciarla en otras sentencias, incluyendo las sentencias de tipo declaración. Veamos un ejemplo:

```
maxi INTEGER := 2*mini;  -- No válido…
mini INTEGER := 15;
```

Sin embargo, PL/SQL **sí que permite la declaración de tipo *forward* para subprogramas**.

Otra restricción de PL/SQL, es que no permite una declaración de este tipo:

```
i, j, k SMALLINT;  -- No válido…
```

Debemos declararlo de la siguiente manera:

```
i SMALLINT;
j SMALLINT;
k SMALLINT;
```

1.3.6 Convenciones de Nomenclatura

En PL/SQL se aplican las mismas convenciones de nomenclatura tanto para los items de los programas como para las unidades, incluyendo esto a las constantes, variables, cursores, cursores con variables, excepciones, procedimientos, funciones y packages. Los nombres pueden ser: **simples, referenciando a un usuario o package (lo llamaremos *qualified*), remotos, o bien uniendo qualified y el hecho de que sea remoto**. Por ejemplo, podemos usar el procedimiento llamado *calcular_salario*, de cualquiera de las siguientes formas:

calcular_salario(…); -- Simple

acciones_emp.calcular_salario(…); -- Qualified

calcular_salario@bbdd_remota(…); -- Remota

acciones_emp.calcular_salario@bbdd_remota(…); -- Qualified y Remota…

En el primer caso, simplemente llamamos al procedimiento que se encuentra en nuestro usuario. En el segundo, utilizamos la notación del punto, puesto que el procedimiento se encuentra almacenado en el Package llamado acciones_emp. En el tercero, llamamos al procedimiento que se encuentra almacenado en una Bbdd remota, a la que hemos llamado bbdd_remota. En último lugar, llamamos a un procedimiento que se encuentra en la bbdd_remota, y además contenido en el package acciones_emp.

1.3.7 Sinónimos

Se pueden crear sinónimos para proporcionar transparencia en el acceso a un esquema remoto de sus tablas, secuencias, vistas, subprogramas y packages. Sin embargo, y como es lógico, no podemos crear sinónimos para los objetos declarados en subprogramas o packages; esto incluye constantes, variables, cursores, cursores con variables, excepciones y procedures de un package (de forma individual).

1.3.8 Ámbito

En el mismo ámbito, todos los identificadores que se declaren deben ser únicos. Por lo tanto, e incluso aunque sus tipos difieran, las variables y parámetros no pueden tener el mismo nombre. Veamos un par de ejemplos:

```
DECLARE
  id_valido BOOLEAN;
  id_valido VARCHAR2(5);  -- No válido, nombre repetido…
  FUNCTION bonus (id_valido IN INTEGER)
    RETURN REAL IS …  -- No válido, nombre repetido dos veces..
```

Veremos en profundidad todo este tema en el apartado dedicado a Visibilidad.

1.3.9 Case Sensitivity

Al igual que para los otros identificadores, los nombres de las constantes, variables y parámetros, **no son case sensitive**. Por ejemplo, PL/SQL considerará iguales a los siguientes nombres:

```
DECLARE
  codigo_postal INTEGER;
  Codigo_postal INTEGER;  -- Igual que el anterior…
  CODIGO_POSTAL INTEGER; -- Igual que los dos anteriores…
```

1.3.10 Resolución de Nombres

Para evitar posibles ambigüedades en sentencias SQL, los nombres de las variables locales y de los parámetros, toman prioridad sobre los nombres de las tablas de Bbdd. Por ejemplo, la siguiente sentencia de UPDATE fallaría, ya que PL/SQL supone que *emp* referencia al contador del loop:

```
FOR emp IN 1..5 LOOP
  …
  UPDATE emp SET bonus = 500 WHERE …
END LOOP;
```

De igual forma, la siguiente sentencia SELECT también fallaría, ya que PL/SQL cree que *emp* referencia al parámetro definido:

```
PROCEDURE calcula_bonus (emp NUMBER, bonus OUT REAL)
IS
  media_sal REAL;
BEGIN
  SELECT AVG(sal) INTO media_sal
```

```
FROM emp WHERE …
```

En estos casos, se debe poner el nombre del usuario de Bbdd antes de la tabla y un punto después… aunque **siempre será más eficiente llamar a la variable o parámetro de otra forma**. Veamos un ejemplo:

```
PROCEDURE calcula_bonus (emp NUMBER, bonus OUT REAL) IS
 media_sal REAL;
BEGIN
 SELECT AVG(sal) INTO media_sal
  FROM usuario.emp WHERE …
```

Al contrario que para el nombre de las tablas, el nombre de las columnas toma prioridad sobre los nombres de las variables locales y parámetros. Por ejemplo, la siguiente sentencia DELETE borrará todos los empleados de la tabla *emp*, y no tan sólo aquellos que se llamen 'Pedro' (que es lo que se pretende), ya que Oracle creerá que los dos *nom_emp* que aparecen en la sentencia WHERE, referencian a la columna de la Bbdd.

```
DECLARE
 nom_emp VARCHAR2(10) := 'Pedro';
BEGIN
 DELETE FROM emp WHERE nom_emp = nom_emp;
```

En estos casos, tenemos dos posibilidades: o bien cambiamos el nombre de la variable (es lo mejor):

```
DECLARE
 mi_nom_emp VARCHAR2(10) := 'Pedro';
```

, o bien utilizamos una label para el bloque:

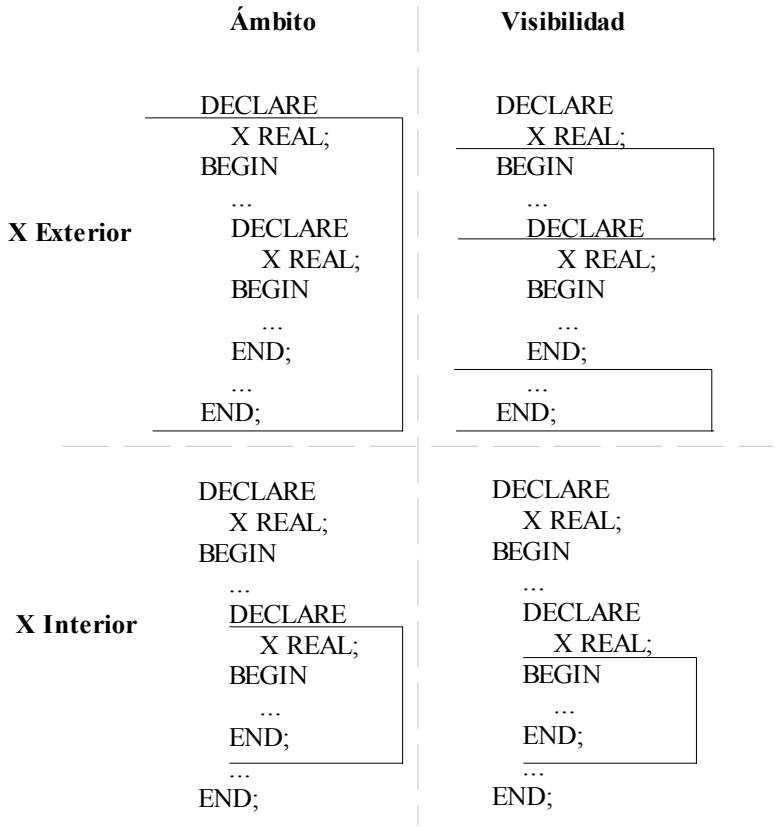
```
<<main>>
DECLARE
 nom_emp VARCHAR2(10) := 'Pedro';
BEGIN
 DELETE FROM emp WHERE nom_emp = main.nom_emp;
```

1.4 Ámbito y Visibilidad

En PL/SQL, las referencias a un identificador son resueltas acorde con su ámbito y su visibilidad. El **ámbito** de un identificador es **aquella parte de una unidad de programa (bloque, subprograma o package), desde la cual se puede referenciar al identificador**. Un identificador es **visible** solo en las partes **desde las que se puede referenciar al identificador, utilizando un nombre adecuado**.

Los identificadores declarados en un bloque PL/SQL, se consideran locales a ese bloque, y globales para todos sus sub-bloques. Si un identificador global es re-declarado en un sub-bloque, ambos identificadores pertenecen al mismo ámbito… sin embargo, en el sub-bloque, tan solo el identificador local es visible porque se debe utilizar un nombre adecuado (*qualified*), para referenciar al global.

Con el siguiente gráfico, se entenderá bien el concepto:



Veamos ahora un ejemplo utilizando unas líneas de código, e indicando que variables son accesibles en cada momento:

```
DECLARE
  a CHAR;
  b REAL;
BEGIN
  -- Identificadores accesibles aquí: a (CHAR), b
  DECLARE
    a INTEGER;
    c REAL;
  BEGIN
    -- Identificadores accesibles aquí: a (INTEGER), b, c
  END;
  …
END;
```

Si quisiéramos referenciar a identificadores del mismo ámbito, pero más externos, deberíamos utilizar etiquetas. Veamos un ejemplo de esto:

```
<<externo>>
DECLARE
  cumple DATE;
BEGIN
  DECLARE
    cumple DATE;
  BEGIN
    …
    IF cumple = exterior.cumple THEN …
```

1.5 Asignaciones

Las variables y constantes se inicializan cada vez que se entra en un bloque o subprograma. Por defecto, las variables se inicializan a NULL, por lo tanto, a menos que se le asigne expresamente, el valor de una variable es **indefinido**. Veamos un caso curioso:

```
DECLARE
  contador INTEGER;
  …
BEGIN
  contador := contador+1;  -- Contador sigue valiendo NULL…
  …
END;
```

Efectivamente, la suma de NULL+1 es siempre NULL, la asignación debe realizarse de manera expresa. En general, **cualquier operación en la cual uno de los operandos sea NULL, nos devolverá un NULL**… es algo que deberemos tener muy en cuenta.

Lo que siga al operador de asignación, puede ser tanto un valor simple (literal numérico), como una expresión compleja… lo único que debemos tener siempre en cuenta es que debe tratarse de un valor del mismo tipo, o por lo menos convertible de forma implícita.

1.5.1 Valores Booleanos

A una variable de tipo booleano, tan sólo le podemos asignar tres valores: TRUE, FALSE y NULL. Por ejemplo, dada la siguiente declaración:

```
DECLARE
  realizado BOOLEAN;
las siguientes sentencias son válidas:
BEGIN
  realizado := FALSE;
  WHILE NOT realizado LOOP
    …
  END LOOP;
  …
```

Cuando se aplica a una expresión, los operadores relacionales, devolverán un valor Booleano. Por tanto, la siguiente asignación será válida:

```
realizado := (cuenta>500);
```

1.5.2 Valores de Base de Datos

De forma alternativa, podemos utilizar la sentencia **SELECT … INTO …** para asignar valores a una variable. Para cada elemento de la lista select, debe existir un tipo compatible en la lista INTO. Veamos un ejemplo:

```
DECLARE
  mi_numemp emp.num_emp%TYPE;
  mi_nomemp emp.nom_emp%TYPE;
  variable NUMBER(7,2);
BEGIN
  …
  SELECT nom_emp, sal+com INTO mi_nomemp, variable
    FROM emp
      WHERE num_emp=mi_numemp;
```

Sin embargo, no podemos seleccionar valores de una columna en una variable de tipo BOOLEAN.

1.6 Expresiones y Comparaciones

Las expresiones se construyen utilizando operandos y operadores. Un **operando** es una variable, constante, literal, o una llamada a una función, que contribuye con un valor a la expresión. Un ejemplo de una expresión aritmética simple sería:

```
-x / 2 + 3
```

Los operadores **unarios**, como por ejemplo la negación (-), actúan sobre un operando, mientras que los **binarios**, como la división, lo hacen sobre dos operandos. PL/SQL no soporta operadores ternarios.

PL/SQL evalúa una expresión mediante la combinación de los valores de los operandos, y la prioridad de los operadores. Veremos esto más en detalle en los siguientes apartados.

1.6.1 Precedencia de los Operadores

Las operaciones en una expresión, son realizadas en un orden particular, dependiendo de la precedencia de los Operadores.

Veamos la tabla de Orden de las Operaciones:

Operador	Operación
**, NOT	Exponenciación, negación lógica
+, -	Identidad, negación
*, /	Multiplicación, división
+, -, ||	Suma, resta, concatenación
=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparación
AND	Conjunción
OR	Disyunción

Los operadores que tienen más prioridad se ejecutan en primer lugar. Cuando los operadores tienen la misma prioridad, se ejecutan en cualquier orden.

Si se desea controlar el orden de ejecución, se deberá utilizar paréntesis para indicarlo.

1.6.2 Operadores Lógicos

Veamos la tabla de verdad de los operadores lógicos, para entender como se evalúan:

x	y	x AND y	x OR y	NOT x
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	
TRUE	NULL	NULL	TRUE	
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	
FALSE	NULL	FALSE	NULL	
NULL	TRUE	NULL	TRUE	NULL
NULL	FALSE	FALSE	NULL	
NULL	NULL	NULL	NULL	

Probablemente, lo que más llamará la atención de esta tabla es que vemos que existen determinadas operaciones entre un valor NULL y otro que no lo es, que sin embargo no dan como resultado NULL, lo cual contradice algo que dijimos en un apartado anterior... esto es cierto, sin embargo Oracle recomienda que **jamás se evalúe una expresión con un operando NULL, ya que potencialmente puede dar un valor indefinido**, y eso debemos tenerlo en cuenta a pesar de esta tabla lógica de verdad.

1.6.3 Operadores de Comparación

Los operadores de comparación, comparan una expresión con otra; el resultado es siempre TRUE, FALSE, o NULL. Normalmente, se utilizarán operadores de comparación en las claúsulas WHERE de una sentencia SQL, y en las sentencias de control condicional.

1.6.3.1 Operadores Relacionales

Los operadores relacionales, nos permitirán comparar expresiones. Veamos la tabla que tiene el significado de cada operador:

Operador	Significado
=	Igual a…
<>, !=, ~=	Diferente a…
<	Menor que…
>	Mayor que…
<=	Menor o igual a…
>=	Mayor o igual a…

Es conveniente la utilización del Operador <>, en lugar de los otros dos para la operación de 'Diferente a…'.

1.6.3.2 Operador IS NULL

El operador IS NULL, devuelve el valor booleano TRUE, si el operando es nulo, o FALSE si no lo es. Es **muy importante utilizar siempre este operador cuando evaluemos si una expresión en nula**, ya que la utilización de una comparación normal, nos daría un valor erróneo. Ejemplo:

IF variable = NULL THEN … -- Jamás debemos utilizarlo

En lugar de esto, debemos escribir…

IF variable IS NULL THEN …

1.6.3.3 Operador LIKE

El operador LIKE se utiliza para comparar un valor alfanumérico con un patrón. En este caso sí que se distinguen las mayúsculas y las minúsculas. LIKE devuelve el valor booleano TRUE, si se produce un 'match' con el patrón, y FALSE si no es así.

Los patrones que podemos 'matchear' con el operador LIKE, pueden incluir dos caracteres especiales llamados *wildcards*. Estos dos caracteres son el 'underscore' (_), y el tanto por ciento (%). El primero nos permitirá que el match devuelva TRUE para un solo carácter cualquiera, mientras que el segundo nos lo permitirá para varios. Lo entenderemos mejor con un ejemplo:

'JUAN' LIKE 'J_AN'; – Devuelve TRUE…

'ANTONIO' LIKE 'AN_IO'; -- Devuelve FALSE…

'ANTONIO' LIKE 'AN%IO'; -- Devuelve TRUE…

1.6.3.4 Operador BETWEEN

El operador BETWEEN, testea si un valor se encuentra en un rango especificado. Su significado literal es 'mayor o igual al valor menor, y menor o igual al valor mayor'. Por ejemplo, la siguiente expresión nos devolvería FALSE:

45 BETWEEN 38 AND 44;

1.6.3.5 Operador IN

El operador IN, comprueba si un valor pertenece a un conjunto. El significado literal sería 'igual a cualquier miembro de…'.

El conjunto de valores puede contener nulos, pero son ignorados. Por ejemplo, la siguiente sentencia **no** borraría los registros de la tabla que tuvieran la columna nom_emp a null.

DELETE FROM emp WHERE nom_emp IN (NULL, 'PEPE', 'PEDRO');

Es más, expresiones del tipo

valor NOT IN conjunto

devolverían FALSE si el conjunto contuviese un NULL. Por ejemplo, en lugar de borrar los registros en los cuales la columna nom_emp fuese distinta de NULL, y diferente de 'PEPE', la siguiente sentencia no borraría nada:

DELETE FROM emp WHERE nom_emp NOT IN (NULL, 'PEPE');

1.6.4 Operador de Concatenación

El operador de concatenación son las dos barras verticales (| |), el cual añade un string a otro. Veamos un ejemplo:

'moto' | | 'sierra' = 'motosierra';

Si ambos operandos son del tipo CHAR, el operador de concatenación devuelve un valor CHAR… en cualquier otro caso devolverá un valor tipo VARCHAR2.

1.7 Funciones Soportadas

PL/SQL proporciona un gran número de funciones bastante potentes para ayudar a manipular la información. Estas funciones pre-definidas se agrupan en las siguientes categorias:

- error-reporting
- numéricas
- carácter
- conversión
- fecha
- misceláneas

Se pueden usar todas las funciones en sentencias SQL excepto las de error-reporting SQLCODE y SQLERRM. También se pueden usar todas las funciones en sentencias de los procedimientos excepto las misceláneas DECODE, DUMP, y VSIZE.

Las funciones de agrupación de SQL: AVG, MIN, MAX, COUNT, SUM, STDDEV, y VARIANCE, no están implementadas en PL/SQL, sin embargo se pueden usar en sentencias SQL (pero no en sentencias de procedimientos).

Veamos una tabla con todas las funciones soportadas por PL/SQL; para una descripción en detalle de cada una de ellas, se puede mirar el Oracle8 SQL Reference.

Error	Numéricas	Carácter	Conversión	Fecha	Misc.
SQLCODE	ABS	ASCII	CHARTOROWID	ADD_MONTHS	DECODE
SQLERRM	ACOS	CHR	CONVERT	LAST_DAY	DUMP
	ASIN	CONCAT	HEXTORAW	MONTHS_BETWEEN	GREATEST
	ATAN	INITCAP	NLS_CHARSET_ID	NEW_TIME	GREATEST_LB
	ATAN2	INSTR	NLS_CHARSET_NAME	NEXT_DAY	LEAST
	CEIL	INSTRB	RAWTOHEX	ROUND	LEAST_UB
	COS	LENGTH	ROWIDTOCHAR	SYSDATE	NVL
	COSH	LENGTHB	TO_CHAR	TRUNC	UID
	EXP	LOWER	TO_DATE		USER
	FLOOR	LPAD	TO_LABEL		USERENV
	LN	LTRIM	TO_MULTI_BYTE		VSIZE
	LOG	NLS_INITCAP	TO_NUMBER		
	MOD	NLS_LOWER	TO_SINGLE_BYTE		
	POWER	NLS_UPPER			
	ROUND	NLSSORT			
	SIGN	REPLACE			
	SIN	RPAD			
	SINH	RTRIM			
	SQRT	SOUNDEX			
	TAN	SUBSTR			
	TANH	SUBSTRB			
	TRUNC	TRANSLATE			
		UPPER			

INNOV@
Desarrollos Informáticos

2 UNIDAD 5:ESTRUCTURAS DE CONTROL

Objetivo general de la unidad

Esta unidad mostrará como estructurar el flujo de control en un programa PL/SQL.

Objetivos específicos

Que el alumno conozca y sepa utilizar en el momento adecuado, todas las estructuras de control aportadas por PL/SQL.

Contenidos

Introducción

Control Condicional

Control Iterativo

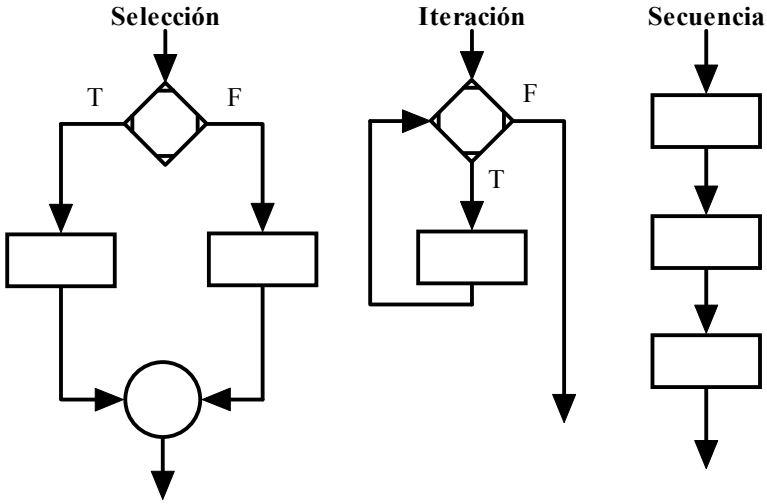
Control Secuencial

INNOV@
Desarrollos Informáticos

Cuaderno de notas

2.1 Introducción

Cualquier programa puede ser escrito utilizando las siguientes estructuras de control básicas.



Se pueden combinar entre sí, para obtener la solución a cualquier problema que se plantee.

2.2 Control Condicional

A menudo, es necesario ejecutar acciones distintas, dependiendo de las circunstancias. Las sentencias IF, nos permiten ejecutar una secuencia de acciones de forma condicional, es decir, el hecho de que se ejecute o no la acción, depende del valor de la condición.

Hay tres variedades de sentencias IF: IF-THEN, IF-THEN-ELSE, y IF-THEN-ELSIF.

2.2.1 IF-THEN

Es la forma más sencilla de una sentencia IF. Asocia una secuencia de sentencias a una condición. Veamos un ejemplo:

IF condicion THEN

secuencia_de_sentencias;

END IF;

La secuencia de sentencias se ejecuta tan sólo si la condición es TRUE.

Se pueden escribir las sentencias IF de forma completa en una sola línea…

IF x > y THEN mayor := x; END IF;

Sin embargo, esto no es bueno ya que no facilita una lectura posterior del código fuente.

2.2.2 IF-THEN-ELSE

Esta segunda variedad añade la palabra clave ELSE, seguida de un conjunto de sentencias. Veamos un ejemplo:

```
IF condicion THEN
  secuencia_1;
ELSE
  secuencia_2;
END IF;
```

Si la condición es TRUE, se ejecutará la secuencia de instrucciones 1, en caso contrario se ejecutará la 2.

Las claúsulas THEN y ELSE pueden incluir sentencias IF, es decir, **podemos agrupar sentencias de tipo IF**. Veamos un ejemplo:

```
IF tipo_transaccion = 'CR' THEN
  UPDATE cuentas SET balance=balance+credito WHERE…
ELSE
  IF nuevo_balance >= balance_minimo THEN
    UPDATE cuentas SET balance=balance-debito WHERE…
  ELSE
    RAISE fondos_insuficientes;
  END IF;
END IF;
```

2.2.3 IF-THEN-ELSIF

Como hemos visto, podemos agrupar sentencias de tipo IF… sin embargo, nos podemos encontrar el caso en que existan muchas posibles alternativas a evaluar, y para cada ELSE tendríamos que abrir una

sentencia de tipo IF-THEN-ELSE, y cerrarla posteriormente… para evitar esto tenemos la palabra clave ELSIF. Veamos un ejemplo:

```
IF condicion1 THEN
  secuencia_1;
ELSIF condicion2 THEN
  secuencia_2;
ELSE
  secuencia_3;
END IF;
```

De esta manera, podemos evaluar tantas como queramos, y sólo deberemos cerrar una sentencia IF con el END IF correspondiente. La última sentencia ELSE se ejecutará cuando no se cumpla ninguna de las anteriores, aunque si no queremos ponerla, pues no pasa nada.

Podemos agrupar tantos ELSIF como deseemos si ningún tipo de problema.

Para entender mejor la utilidad de ELSIF, vamos a ver dos ejemplos, en el primero programaríamos las condiciones utilizando sentencias IF-THEN-ELSE normales, mientras que en el segundo utilizaríamos ELSIF… así podremos apreciar realmente lo que ganamos en cuanto a comodidad y a facilidad de lectura e interpretación posterior.

```
IF condicion1 THEN
  Sentencia1;
ELSE
   IF condicion2 THEN
     Sentencia2;
   ELSE
      IF condicion3 THEN
        Sentencia3;
      END IF;
   END IF;
END IF;
```

```
IF condicion1 THEN
  Sentencia1;
ELSIF condicion2 THEN
  Sentencia2;
ELSIF condicion3 THEN
  Sentencia3;
END IF;
```

2.3 Control Iterativo

Las sentencias de tipo LOOP, permiten ejecutar una secuencia de sentencias múltiples veces.

Existen tres variedades de sentencias LOOP: LOOP, WHILE-LOOP, y FOR-LOOP.

2.3.1 LOOP

Se trata de la variedad más simple de la sentencia LOOP, y se corresponde con el bucle básico (o infinito), el cual incluye una secuencia de sentencias entre las palabras claves LOOP y END LOOP. Veamos un ejemplo

```
LOOP
  secuencia_de_sentencias;
END LOOP;
```

En cada iteración del bucle, se ejecutan todas las sentencias de forma secuencial. Evidentemente es raro que deseemos tener un bucle infinito en un programa, por tanto existe una manera de forzar la salida, y es la utilización de la palabra clave EXIT. Para esta palabra también tenemos dos variedades posibles: EXIT y EXIT-WHEN. Vamos a verlas en detalle.

2.3.1.1 EXIT

La sentencia EXIT provoca la salida de un bucle de forma incondicional. Cuando se encuentra un EXIT, el bucle acaba inmediatamente y el control pasa a la siguiente instrucción que esté fuera del bucle. Veamos un ejemplo:

```
LOOP
  …
  IF limite_credito < 3 THEN
    …
    EXIT;  -- Fuerza la salida inmediata…
  END IF;
END LOOP;
-- El control pasaría a esta instrucción…
```

Veamos ahora un ejemplo donde no podemos utilizar EXIT:

```
BEGIN
 …
 IF limite_credito < 3 THEN
  …
  EXIT; -- Sentencia no válida aquí…
 END IF;
END;
```

La sentencia EXIT siempre debe encontrarse dentro de un bucle LOOP. Para salir de un bloque PL/SQL que no sea un bucle antes de su finalización normal, podemos usar la sentencia RETURN.

2.3.1.2 EXIT-WHEN

La sentencia EXIT-WHEN, nos va a permitir salir de un bucle de forma condicional. Cuando PL/SQL encuentra una sentencia de este tipo, la condición del WHEN será evaluada… en caso de devolver TRUE, se provocará la salida del bucle… en caso contrario, se continuará la iteración. Veamos un ejemplo:

```
LOOP
 FETCH c1 INTO …
 EXIT WHEN c1%NOTFOUND;  -- Salir si se cumple la condición
 …
END LOOP;
CLOSE c1;
```

De forma parecida a lo que ocurría con los IF y los ELSIF, también podríamos controlar la salida de un bucle de otra forma, y no mediante un EXIT-WHEN… lo que ocurre es que, al igual que en el caso anterior, la utilización de esta sentencia facilitará la programación y lectura de nuestro código. Veamos un ejemplo:

```
IF contador>100 THEN
  EXIT;
END IF;
```

```
EXIT WHEN contador>100;
```

2.3.1.3 Etiquetas de los bucles

Al igual que los bloques de PL/SQL, los bucles pueden ser etiquetados. La etiqueta es un identificador no declarado que se escribe entre los símbolos << y >>; deben aparecer al principio de las sentencias LOOP. Veamos un ejemplo:

```
<<nombre_etiqueta>>
LOOP
  secuencia_de_sentencias;
END LOOP;
```

De forma opcional, y para facilitar la lectura del código, el nombre de la etiqueta puede aparecer también al final de la sentencia LOOP. Veamos el ejemplo:

```
<<mi_loop>>
LOOP
  secuencia_de_sentencias;
END LOOP mi_loop;
```

Utilizando etiquetas y la sentencia EXIT, podemos forzar la salida no sólo de un bucle, sino de cualquiera que esté incluido en el etiquetado. Veamos un ejemplo de esto:

```
<<exterior>>
LOOP
  …
  LOOP
    …
    EXIT exterior WHEN …  -- Sale de los dos bucles LOOP…
  END LOOP;
  …
END LOOP exterior;
```

De manera general, saldría de cualquier bucle que fuera interior al bucle etiquetado. Esto puede ser muy útil en determinadas circunstancias.

2.3.2 WHILE-LOOP

La sentencia WHILE-LOOP, asocia una condición a una secuencia de instrucciones que se encuentran entre las palabras claves LOOP y END LOOP. Veamos un ejemplo:

```
WHILE condicion LOOP
  secuencia_de_instrucciones;
END LOOP;
```

Antes de cada iteración del LOOP, la condición se evalúa… si devuelve TRUE se continúa iterando, en caso de que devuelva FALSE o NULL, se forzará la salida del bucle.

El número de iteraciones depende de la condición, y es desconocido hasta que el bucle termina. Puede haber 0 o N iteraciones hasta que la condición sea FALSE.

Algunos lenguajes tienen estructuras como LOOP UNTIL, o REPEAT UNTIL, las cuales evalúan la condición al final y no al principio de todo. PL/SQL no tiene esta estructura, sin embargo sería muy fácil simularla. Veamos un ejemplo:

```
LOOP
  secuencia_de_instrucciones;
  EXIT WHEN expresion_booleana;
END LOOP;
```

Para asegurarnos que un bucle de tipo WHILE se ejecuta por lo menos una vez, podemos implementarlo mediante una variable booleana de la siguiente forma:

```
hecho:=FALSE;
WHILE NOT hecho LOOP
  secuencia_de_instrucciones;
  hecho:=expresion_booleana;
END LOOP;
```

2.3.3 FOR-LOOP

Al contrario que en el caso de un bucle WHILE, en el cual recordemos que el número de iteraciones era desconocido a priori, en un bucle FOR este número es **conocido antes de comenzar la iteración**. Los bucles FOR iteran un número de veces que está comprendido en un rango. Veamos la sintaxis mediante un ejemplo:

```
FOR contador IN [REVERSE] valor_minimo..valor_maximo LOOP
  secuencia_de_instrucciones;
END LOOP;
```

El rango es evaluado cuando se entra por primera vez en el bucle, y nunca más se vuelve a evaluar.

Vemos unos cuantos ejemplos que pongan de manifiesto la utilización del bucle FOR:

```
FOR i IN 1..3 LOOP  -- Asigna los valores 1, 2, 3 a i
  secuencia_de_instrucciones;  -- Se ejecutan tres veces…
END LOOP;
FOR i IN 3..3 LOOP – Asigna el valor 3 a i
 secuencia_de_instrucciones;  -- Se ejecutan una vez…
END LOOP;
FOR i IN REVERSE 1..3 LOOP  -- Asigna los valores 3, 2, 1 a i
  secuencia_de_instrucciones;  -- Se ejecutan tres veces…
END LOOP;
```

Dentro de un bucle FOR, el contador del bucle puede ser referenciado como una constante… por lo tanto, el contador puede aparecer en expresiones, pero **no se le puede asignar ningún valor**. Veamos un ejemplo de esto:

```
FOR ctr IN 1..10 LOOP
  IF NOT fin THEN
    INSERT INTO … VALUES (ctr, …); -- Válido…
    factor:=ctr*2;  -- Válido…
  ELSE
    ctr:=10; -- No válido…
  END IF;
END LOOP;
```

2.3.3.1 Esquemas de Iteración

Los rangos de un bucle FOR pueden ser literales, variables, o expresiones, pero deben **poder ser siempre evaluadas como enteros**. Por ejemplo, los siguientes esquemas de iteración son legales:

j IN –5..5

k IN REVERSE primero..ultimo

*step IN 0..TRUNC(mayor/menor)*2*

codigo IN ASCII('A')..ASCII('J')

Como podemos apreciar, el valor menor no es necesario que sea 1; sin embargo, el incremento (o decremento) del contador del bucle debe ser 1. Algunos lenguajes proporcionan una claúsula STEP, la cual permite especificar un incremento diferente. Veamos un ejemplo codificado en BASIC:

```
FOR J = 5 TO 15 STEP 5  :REM Asigna valores 5,10,15 a J
  secuencia_de_instrucciones  -- J tiene valores 5,10,15
NEXT J
```

PL/SQL no soporta ninguna estructura de este tipo, sin embargo podemos simular una de manera muy sencilla. Veamos como haríamos lo anterior utilizando PL/SQL:

```
FOR j IN 5..15 LOOP  -- Asigna los valores 5,6,7,.. a j
  IF MOD(j,5)=0 THEN  -- Solo pasan los múltiplos de 5…
    secuencia_de_instrucciones;  -- j tiene valores 5,10,15
  END IF;
END LOOP;
```

2.3.3.2 Rangos dinámicos

PL/SQL permite determinar el rango del LOOP de forma dinámica en tiempo de ejecución. Veamos un ejemplo:

```
SELECT COUNT(num_emp) INTO cont_emp FROM emp;
FOR i IN 1..num_emp LOOP
  …
END LOOP;
```

El valor de num_emp es desconocido cuando se compila… es en tiempo de ejecución cuando se le asigna un valor.

Cuando el valor mínimo es mayor al máximo existente en un bucle FOR, lo que ocurre es que el bucle **no se ejecutará ninguna vez**. Ejemplo:

```
-- limite vale 1
FOR i IN 2..limite LOOP
  secuencia_de_instrucciones;  -- Se ejecutan 0 veces…
END LOOP;
-- El control pasa aquí…
```

2.3.3.3 Reglas de Ámbito y Visibilidad

El contador de un bucle se define tan sólo para el bucle, no se puede referenciar desde fuera del mismo. Después de terminar el bucle, el contador es indefinido. Veamos un ejemplo:

```
FOR ctr IN 1..10 LOOP
  …
END LOOP;
sum:=ctr-1;  -- Sentencia No Válida…
```

No es necesario declarar de forma explícita el contador de un bucle, ya que al utilizarlo se declara de forma implícita como una variable local de tipo INTEGER. En el siguiente ejemplo veremos como la declaración local anula cualquier declaración global:

```
DECLARE
  ctr INTEGER;
BEGIN
  …
  FOR ctr IN 1..25 LOOP
    …
    IF ctr>10 THEN …  -- Referenciará al contador del bucle…
  END LOOP;
END;
```

Para referenciar a la variable global en el ejemplo anterior, se debe usar una etiqueta. Veamos como hacerlo:

```
<<principal>>
DECLARE
  ctr INTEGER;
BEGIN
  …
  FOR ctr IN 1..25 LOOP
    …
    IF principal.ctr>10 THEN …  -- Referenciará a la variable global…
  END LOOP;
END principal;
```

2.4 Control Secuencial

Al contrario que las sentencias IF y LOOP, las instrucciones GOTO y NULL (que son las asociadas al control secuencial), no son cruciales ni imprescindibles dentro de la programación en PL/SQL. La estructura del PL/SQL es tal, que la sentencia GOTO **no es necesaria de forma obligatoria**. De todas formas, en algunas ocasiones, puede estar justificado su uso para simplificar un problema.

El uso de la sentencia NULL, puede ayudar a la comprensión de un programa, puesto que en una sentencia de tipo condicional indicaría que en un determinado caso no hay que hacer nada.

Sin embargo, el uso de sentencias GOTO si que puede ser más catastrófico, ya que pueden provocar un *código complejo, y no estructurado*, que es difícil de entender y mantener. Por lo tanto, **solo hay que emplear GOTO cuando esté fuertemente justificado**. Por ejemplo, cuando se desee salir de una estructura profunda (agrupación de bucles) a una rutina de manejo de errores, entonces se podría utilizar la sentencia GOTO.

2.4.1 Sentencia GOTO

La sentencia GOTO salta a una etiqueta de forma incondicional; la etiqueta debe ser única en su ámbito, y **debe preceder a una sentencia ejecutable, o a un bloque PL/SQL**. Cuando se ejecuta, la sentencia GOTO transfiere el control a la sentencia o bloque etiquetados. Veamos un ejemplo:

```
BEGIN
  …
  GOTO insercion_fila;
  …
  <<insercion_fila>>
  INSERT INTO emp VALUES …
END;
```

En el ejemplo anterior, hemos visto un salto 'hacia abajo'… veamos ahora un salto 'hacia arriba':

```
BEGIN
  …
  <<actualizar_fila>>
  BEGIN
    UPDATE emp SET …
    …
  END;
  …
  GOTO actualizar_fila;
  …
END;
```

Como hemos dicho, una etiqueta debe preceder a una sentencia ejecutable; veamos un ejemplo que no funcionaría debido a esta circunstancia:

```
DECLARE
  hecho BOOLEAN;
BEGIN
  …
  FOR i IN 1..50 LOOP
    IF hecho THEN
      GOTO fin_loop;
    END IF;
    …
```

```
<<fin_loop>>  -- No válido…
END LOOP;  -- Sentencia No ejecutable…
END;
```

Podríamos solucionarlo tan sólo incluyendo una sentencia ejecutable después de la etiqueta, como por ejemplo NULL. Vamos a ver como lo haríamos:

```
DECLARE
 hecho BOOLEAN;
BEGIN
 …
 FOR i IN 1..50 LOOP
  IF hecho THEN
   GOTO fin_loop;
  END IF;
  …
 <<fin_loop>>
 NULL;  -- Sentencia ejecutable…
 END LOOP;
END;
```

2.4.1.1 Restricciones

Algunos destinos en un salto de tipo GOTO no están permitidos. De forma específica, una sentencia GOTO no puede saltar a:

- Una sentencia IF
- Una sentencia LOOP
- Un Sub-Bloque
- Fuera de un Sub-Programa

2.4.2 Sentencia NULL

La sentencia NULL, cuando se emplea sola, sin asignarla a nada, especifica literalmente **'ninguna acción'**. Tiene dos utilidades principalmente, una es la de clarificar el código fuente para aquellos casos en los que el programa no debe hacer nada, como por ejemplo en una sentencia IF-THEN-ELSE. Veamos un ejemplo:

```
IF contador>0 THEN
  -- Hacemos algo…
ELSE
  NULL;
END IF;
```

Otra utilidad es cuando queramos hacer un 'debug' de alguna parte del programa, ya que podemos compilar una parte del mismo, y en la parte que no hayamos programado todavía, podemos poner un NULL… de hecho debemos hacerlo, ya que sino no funcionaría. Veamos un ejemplo:

```
IF num_emp>500 THEN
  -- Parte grande que queremos probar…
ELSE
END IF;
```

Si intentamos compilar esto, no funcionaría… deberíamos poner lo siguiente:

```
IF num_emp>500 THEN
  -- Parte grande que queremos probar…
ELSE
  NULL;
END IF;
```

Así si que iría bien.

3 UNIDAD 6:INTERACCIÓN CON ORACLE

Objetivo general de la unidad

Clarificar como PL/SQL interactua con la Bbdd Oracle, permitiendo que el lenguaje sea una extensión procedimental del SQL propietario de Oracle.

Objetivos específicos

Que el alumno conozca como PL/SQL soporta las funciones de SQL, permite SQL dinámico, y que además sepa definir y controlar cursores sobre Bbdd.

Contenidos

Soporte SQL

Manejando Cursores

Empaquetando Cursores

Utilización de Cursores con bucles FOR

INNOV@
Desarrollos Informáticos

Cuaderno de notas

3.1 Soporte SQL

Al ser una extensión del SQL, PL/SQL ofrece una combinación única de potencia y facilidad de uso. Se puede tratar toda la información de Oracle con una gran flexibilidad y seguridad, ya que PL/SQL soporta todos los comandos de manipulación de datos, de control de transacciones, funciones, pseudocolumnas, y operadores. PL/SQL también cumple los estándares SQL92, y ANSI/ISO.

PL/SQL no soporta comandos de definición de datos, como ALTER y CREATE.

3.1.1 Manipulación de datos

Para manipular datos en Oracle, utilizaremos los comandos ya conocidos de INSERT, UPDATE, DELETE, SELECT y LOCK TABLE.

3.1.2 Control de transacciones

Oracle está orientado a la transacción, es decir, Oracle utiliza las transacciones para asegurar la integridad de la Bbdd. Una *transacción* es un conjunto de sentencias de manipulación de datos que se comportan como una sola unidad lógica de trabajo.

No entraremos a fondo en el concepto de transacciones, ni de las sentencias que se utilizan, ya que deben ser conceptos conocidos por los alumnos de este curso.

Sin embargo, sí diremos que PL/SQL utiliza los comandos COMMIT, ROLLBACK, SAVEPOINT y SET TRANSACTION, para realizar el control de las transacciones.

3.1.3 Funciones SQL

PL/SQL permite utilizar todas las funciones de SQL, incluyendo las siguientes funciones de agrupación: AVG, COUNT, MAX, MIN, STDDEV, SUM, y VARIANCE.

Se pueden utilizar las funciones de agrupación en sentencias SQL, **pero no como sentencias individuales en los procedimientos**.

Al igual que en el apartado anterior, no vamos a entrar en detalle acerca del uso que tienen las funciones, puesto que se sale del objetivo del curso.

3.1.4 Pseudocolumnas de SQL

PL/SQL reconoce las siguientes pseudocolumnas de SQL, las cuales devuelven una determinada información específica de los datos: CURRVAL, LEVEL, NEXTVAL, ROWID, y ROWNUM.

Las pseudocolumnas no son columnas reales de las tablas de la Bbdd, pero se comportan como tales. Por ejemplo, se pueden seleccionar valores de una pseudocolumna, pero no se pueden insertar, actualizar o borrar valores de la misma. Además, estas pseudocolumnas son comunes a todas las tablas de la Bbdd… son columnas lógicas.

Se pueden utilizar pseudocolumnas en sentencias SQL, pero no como elementos individuales de un programa PL/SQL.

Vamos a dar una breve descripción de las pseudocolumnas.

3.1.4.1 CURRVAL y NEXTVAL

Como ya sabemos, una secuencia es un objeto de un esquema, que genera números secuenciales. Cuando se crea una secuencia, se puede especificar su valor inicial y su incremento.

CURRVAL devuelve el valor actual de la secuencia que se especifique. Veamos un ejemplo:

SELECT nombre_secuencia.CURRVAL FROM sys.dual;

NEXTVAL devuelve el siguiente valor de una secuencia que se le especifique… hay que tener en cuenta que además de hacerlo, **actualizará el nuevo valor dentro de la secuencia**, ya que se trata de una sentencia que realiza el COMMIT de forma implícita. Esto es debido a que de esta forma soluciona el problema del acceso concurrente a la secuencia. Veamos un ejemplo:

Valor actual de mi_secuencia=3

SELECT mi_secuencia.NEXTVAL FROM sys.dual;

Devuelve 4, y además el valor actual de mi_secuencia=4.

3.1.4.2 LEVEL

Se utiliza LEVEL con la sentencia SELECT CONNECT BY, para organizar los registros de una tabla de Bbdd en una estructura tipo árbol. LEVEL devuelve el número de nivel de un nodo en una estructura tipo árbol. El nivel raíz es 1, sus hijos son de nivel 2, sus nietos de nivel 3, y así sucesivamente…

3.1.4.3 ROWID

ROWID devuelve el *rowid* (dirección binaria física) de una fila en una tabla de la Bbdd. Recordemos que PL/SQL tiene un tipo de datos llamado ROWID; se puede utilizar variables de este tipo, para almacenar los *rowids* en un formato legible.

Cuando se selecciona una *rowid* dentro de un tipo ROWID, se debe usar la función ROWIDTOCHAR, la cual convertirá el valor binario a un string de caracteres de 18-bytes. Una vez hecho esto, podremos comparar el valor de una variable ROWID con la pseudocolumna ROWID en una claúsula WHERE de una sentencia UPDATE o DELETE, para identificar el último registro recuperado en un cursor (Es un ejemplo).

3.1.4.4 ROWNUM

ROWNUM devuelve el número lógico que nos indica el orden en el cual una fila ha sido recuperada de una tabla por una sentencia SELECT.

3.1.5 Operadores SQL

PL/SQL permite utilizar todos los comandos de comparación, conjunto, y operaciones sobre filas de SQL. No vamos a entrar en detalle en ellos, puesto que su funcionamiento es idéntico al ya conocido en SQL. Entre estos operadores podemos mencionar: ALL, ANY, SOME, BETWEEN, EXISTS, IN, IS NULL, LIKE, etc…

3.1.6 Utilizando DDL y SQL Dinámico

En este apartado explicaremos el por qué PL/SQL no acepta el lenguaje de definición de datos de SQL, y daremos un apunte de cómo resolverlo utilizando SQL Dinámico.

3.1.7 Eficiencia versus Flexibilidad

Antes de que un programa PL/SQL sea ejecutado, éste debe ser compilado. El compilador de PL/SQL resuelve las referencias a objetos de un esquema de Oracle, mirando sus definiciones en el diccionario de datos; una vez hecho esto, el compilador asigna el espacio necesario a las variables del programa, de forma que éste pueda ejecutarse posteriormente. Este proceso se llama *binding*.

La forma en que un lenguaje de acceso a Bbdd implementa el *binding*, afecta a la eficiencia en tiempo de ejecución y a la flexibilidad. Hacer el *binding* en tiempo de compilación (llamado *binding estático o early binding*), incrementa la eficiencia, ya que las definiciones de los objetos de los esquemas son comprobadas en ese momento, y no en tiempo de ejecución. Por otro lado, hacer el *binding* en tiempo de ejecución (llamado *binding dinámico o late binding*), incrementa la flexibilidad ya que las definiciones de los objetos del esquema son desconocidas hasta el momento de la ejecución.

PL/SQL fue diseñado para el procesamiento de transacciones a alta velocidad, y por ello incrementa el rendimiento mediante la utilización del *early binding*. Al contrario que el SQL, el cual es compilado y ejecutado sentencia a sentencia en tiempo de ejecución (*late binding*), PL/SQL se procesa obteniendo un código compilado llamado *p_code* (esta es la técnica de *early binding*). En tiempo de ejecución, el motor de PL/SQL simplemente ejecuta este código compilado.

3.1.8 Algunas Limitaciones

La técnica escogida por los diseñadores del PL/SQL, tiene asociada algunas limitaciones. Por ejemplo, el *p_code* incluye referencias a objetos de esquema como tablas y procedimientos almacenados; el compilador de PL/SQL puede resolver las referencias a dichos objetos tan sólo en el caso de que sean conocidos en tiempo de compilación. En el siguiente ejemplo, el compilador no puede procesar el procedimiento, porque la tabla no está definida hasta que el procedimiento se ejecuta.

```
CREATE PROCEDURE crear_tabla AS
BEGIN
  CREATE TABLE dept (num_dept NUMBER(2), …); -- No válido…
  …
END;
```

Veamos otro ejemplo en el que el compilador tampoco puede hacer un *bind* de la referencia a la tabla en la sentencia DROP TABLE, ya que el nombre de la tabla es desconocido hasta que el procedimiento se ejecuta:

```
CREATE PROCEDURE drop_tabla (nombre_tabla IN VARCHAR2)
AS
BEGIN
  DROP TABLE nombre_tabla; -- No válido…
  …
END;
```

3.1.9 Evitando las limitaciones

Sin embargo, el package DBMS_SQL, proporcionado por Oracle, permite a PL/SQL ejecutar sentencias de definición y manipulación de datos de forma dinámica **en tiempo de ejecución**. Veamos un ejemplo en el cual definimos un procedimiento almacenado que cuando sea llamado, efectuará un drop de una tabla de la Bbdd que se le especifique:

```
CREATE PROCEDURE drop_tabla (nombre_tabla IN
VARCHAR2) AS
  id_cursor INTEGER;
BEGIN
  /* Abrimos un nuevo cursor y devuelve su identificador */
  id_cursor := DBMS_SQL.OPEN_CURSOR;
  /* Efectuamos un Parse de la sentencia, en la cual concatenaremos un
DROP TABLE, con el nombre de la tabla */
  DBMS_SQL.PARSE(id_cursor, 'DROP TABLE '||nombre_tabla,
dbms_sql.native);
  /* La ejecutamos */
  DBMS_SQL.EXECUTE;
  /* Cerramos el Cursor */
```

```
  DBMS_SQL.CLOSE_CURSOR(id_cursor);
EXCEPTION
  /* Si se ejecuta la excepción, cerramos el cursor antes de salir */
  WHEN OTHERS THEN
    DBMS_SQL.CLOSE_CURSOR(id_cursor);
END drop table;
```

El SQL dinámico tiene mucha más potencia, y permite la ejecución de sentencias que pueden ser simples, o incluso funciones y procedimientos parametrizados. No es el objetivo del curso entrar en detalle sobre el funcionamiento del SQL dinámico, puesto que es un tema bastante más extenso y complicado. Una buena referencia sobre su funcionamiento lo podemos encontrar en el libro *Oracle 8 Application Developer's Guide.*

3.2 Manejando Cursores

El conjunto de filas que devuelve una consulta, puede ser 0, 1 o N, dependiendo de a cuantas filas afecte la condición de búsqueda. Cuando una consulta devuelve múltiples filas, se debe declarar un cursor para procesarlas. Se puede declarar un cursor en la parte de declaraciones de cualquier bloque, subprograma o package PL/SQL.

Se utilizan tres instrucciones para controlar un cursor: OPEN, FETCH, y CLOSE. En primer lugar, se inicializa el cursor con la sentencia OPEN, la cual identifica al conjunto resultante. Entonces, se usa la sentencia FETCH para recuperar la primera fila; se puede ejecutar FETCH de manera repetida hasta que todas las filas han sido tratadas, cuando se procesa la última, se debe cerrar el cursor con la sentencia CLOSE. Se pueden procesar varias consultas de forma paralela, declarando y abriendo múltiples cursores.

3.2.1 Declaración de un Cursor

Las declaraciones de tipo *forward* no se permiten en PL/SQL, por lo tanto, se debe declarar un cursor **antes de referenciarlo en otras sentencias**. Cuando se declara un cursor, se debe ponerle un nombre y asociarlo con una consulta específica utilizando la siguiente sintaxis:

```
CURSOR nombre_cursor [ (parametro[, parametro] … )]
  [RETURN tipo_que_devuelve] IS sentencia_select;
```

donde *tipo_que_devuelve*, representa un registro o fila de una tabla de Bbdd, y *parametro* tiene la siguiente sintaxis:

nombre_parametro_cursor [IN] tipo_de_dato [{:= | DEFAULT} expresion]

Por ejemplo, podemos declarar dos cursores llamados c1 y c2, de la siguiente forma:

```
DECLARE
  CURSOR c1 IS
    SELECT num_emp, nom_emp, trab_emp, sal_emp FROM emp
      WHERE sal_emp>2000;
  CURSOR c2 RETURN dept%ROWTYPE IS
    SELECT * FROM dept WHERE num_dept=10;
```

El nombre de un cursor, es un identificador que no debe ser declarado, no el nombre de una variable de PL/SQL. No se pueden asignar valores a un nombre de un cursor, ni usarlo en una expresión. Sin embargo, los cursores y las variables **siguen las mismas reglas de ámbito y visibilidad**. Se permite dar a un cursor el mismo nombre que una tabla de Bbdd… sin embargo se recomienda no hacerlo.

Un cursor puede tener parámetros, los cuales deben aparecer en la consulta asociada. Los parámetros son de tipo IN, ya que no pueden devolver ningún valor. Tampoco podemos imponer la constraint de NOT NULL a un parámetro de un cursor.

Veamos un ejemplo de definición de un cursor, en la cual asignamos valores por defecto a los parámetros de dicho cursor:

```
DECLARE
  CURSOR c1 (Minimo INTEGER DEFAULT 0,
             Maximo INTEGER DEFAULT 99) IS
    SELECT …
```

El ámbito de los parámetros de un cursor es local al cursor, es decir, que pueden ser referenciados solo en la consulta especificada en la declaración del cursor. Los valores de los parámetros del cursor son usados por la consulta asociada cuando se abre el cursor.

3.2.2 Apertura de un Cursor

Cuando se abre un cursor, se ejecuta la consulta que tiene asociada, la cual recupera todas las filas que se ven incluidas en su condición de búsqueda. Para aquellos cursores que han sido declarados utilizando la claúsula FOR UPDATE, la sentencia OPEN también se encarga de bloquear las filas. Veamos un ejemplo de sentencia OPEN:

```
DECLARE
 CURSOR c1 IS
  SELECT nom_emp, trab_emp
   FROM emp
    WHERE sal_emp<3000;
 …
BEGIN
 OPEN c1;
 …
END;
```

Las filas del conjunto resultante no son recuperadas cuando se ejecuta la sentencia OPEN, sino que lo serán más tarde cuando se ejecute la sentencia FETCH.

3.2.2.1 Paso de parámetros

También se utiliza la sentencia OPEN para pasar parámetros a un cursor. A menos que se deseen aceptar los valores establecidos por defecto, cada parámetro formal del cursor debe tener un parámetro correspondiente en la sentencia OPEN. Veamos un ejemplo:

```
DECLARE
 nombre emp.nom_emp%TYPE;
 salario emp.sal_emp%TYPE;
 CURSOR c1 (nom VARCHAR2, sueldo NUMBER) IS
  SELECT …
```

Cualquiera de las siguientes sentencias abriría el cursor:

```
OPEN c1 (nombre, 3000);
OPEN c1 ('FRANCISCO', 4000);
OPEN c1 (nombre, salario);
```

3.2.3 Recuperación de valores de un Cursor

La sentencia FETCH **recupera una a una, las filas resultantes de la apertura del cursor**. Después de cada *fetch*, el cursor avanza a la siguiente fila de la consulta. Veamos un ejemplo:

```
FETCH c1 INTO mi_numemp, mi_nomemp, mi_numdept;
```

Para cada valor de columna devuelto por la consulta asociada al cursor, debe existir su correspondiente variable en la lista INTO. Obviamente, sus tipos de datos deben ser compatibles. Normalmente, se emplea la sentencia FETCH de la siguiente manera:

```
LOOP
  FETCH c1 INTO mi_record;
  EXIT WHEN c1%NOTFOUND;
  -- Procesa la información del record…
END LOOP;
```

La consulta puede referenciar a variables de PL/SQL que estén en su ámbito. Sin embargo, algunas variables de la consulta se evalúan tan sólo cuando se abre el cursor. En el siguiente ejemplo que veremos, cada salario recuperado se multiplica por 2, aunque *factor* se vaya incrementando después de cada *fetch*:

```
DECLARE
  mi_sal emp.sal_emp%TYPE;
  mi_trabajo emp.trab_emp%TYPE;
  factor INTEGER:=2;
  CURSOR c1 IS
    SELECT factor*sal_emp FROM emp
      WHERE trab_emp=mi_trabajo;
BEGIN
  …
  OPEN c1;  -- Aquí factor vale 2…
  LOOP
    FETCH c1 INTO mi_sal;
    EXIT WHEN c1%NOTFOUND;
    factor:=factor+1;  -- No Afecta al Fetch…
  END LOOP;
```

```
END;
```

Para cambiar el resultado del conjunto, o los valores de las variables de la consulta, se debe cerrar y abrir el cursor con las variables conteniendo los nuevos valores.

Se puede utilizar un lista INTO diferente en *fetches* separados para el mismo cursor. Veamos un ejemplo:

```
DECLARE
  CURSOR c1 IS
    SELECT nom_emp FROM emp;
  nombre1 emp.nom_emp%TYPE;
  nombre2 emp.nom_emp%TYPE;
  nombre2 emp.nom_emp%TYPE;
BEGIN
  OPEN c1;
  FETCH c1 INTO nombre1;  -- Recupera la primera fila…
  FETCH c1 INTO nombre2;  -- Recupera la segunda fila…
  FETCH c1 INTO nombre3;  -- Recupera la tercera fila…
  …
  CLOSE c1;
END;
```

Si se realiza un *fetch* después de haber recuperado la última fila de la consulta, los valores de las variables son indeterminados.

3.2.4 Cierre de un Cursor

La sentencia CLOSE deshabilita el cursor, y el resultado del mismo pasa a ser indefinido. Un ejemplo de la sentencia CLOSE sería:

```
CLOSE c1;
```

Una vez que un cursor se cierra, se puede volver a abrir sin problemas. Cualquier otra operación sobre un cursor cerrado, *provoca una excepción que se llama INVALID_CURSOR.*

3.2.5 Utilización de Subconsultas

Una *subconsulta* es una consulta (normalmente entre paréntesis), que aparece **dentro de otra sentencia de manipulación de datos**. Cuando la subconsulta se evalúa, proporciona un conjunto de valores a la otra sentencia. Normalmente, las subconsultas se utilizan en las claúsulas WHERE. En el siguiente ejemplo, la consulta devuelve los empleados que no viven en Burgos:

```
DECLARE
 CURSOR c1 IS
  SELECT num_emp, nom_emp FROM emp
   WHERE num_dept IN (SELECT num_dept FROM dept
       WHERE loc_dept<>'BURGOS');
```

Utilizando una subconsulta en la claúsula FROM, la siguiente consulta devolverá el número y nombre de cada departamento que tenga cinco o más empleados:

```
DECLARE
 CURSOR c1 IS
  SELECT t1.num_dept, nom_dept, "STAFF"
   FROM dept t1, (SELECT num_dept, COUNT(*) "STAFF"
    FROM emp GROUP BY num_dept) t2
   WHERE t1.num_dept = t2.num_dept AND "STAFF">=5;
```

3.3 Empaquetando Cursores

Es posible separar la especificación del cursor de su cuerpo en un Package. De esta forma, se puede cambiar el cuerpo del cursor sin tener que cambiar la especificación. Se codifica la especificación del cursor en la parte declarativa del package utilizando la siguiente sintaxis:

```
CURSOR nombre_cursor [ (parametro [, parametro] … ) ]
 RETURN tipo_de_retorno;
```

En el siguiente ejemplo, utilizamos el atributo %ROWTYPE para proporcionar un tipo registro, que representa a una fila en la tabla de Bbdd emp:

```
CREATE PACKAGE acciones_emp AS
  /* Declaramos la especificación del Cursor */
  CURSOR c1 RETURN emp%ROWTYPE;
  …
END acciones_emp;

CREATE PACKAGE BODY acciones_emp AS
  /* Definimos el cuerpo del Cursor */
  CURSOR c1 RETURN emp%ROWTYPE IS
    SELECT * FROM emp
      WHERE sal_emp>3000;
  …
END acciones_emp;
```

La especificación del cursor no tiene sentencia SELECT, ya que la claúsula RETURN define el tipo de dato del valor resultante. Sin embargo, el cuerpo del cursor debe tener la sentencia SELECT, y la misma claúsula RETURN que la de su especificación. También deben coincidir el número y tipos de datos de los elementos seleccionados con el SELECT, con la claúsula RETURN.

Los cursores empaquetados incrementan la flexibilidad, ya que se puede cambiar el cuerpo del cursor, sin tener que modificar su definición.

3.4 Utilización de Cursores con bucles FOR

En la mayoría de los casos prácticos, **podemos sustituir las instrucciones de manejo de cursores: OPEN, FETCH y CLOSE, por la utilización de bucles FOR**. Esto simplifica mucho el funcionamiento ya que no es necesario abrir el cursor, ni realizar el fetch, ni cerrarlo posteriormente, ya que todas estas operaciones van implícitas cuando se usa el bucle FOR. Veamos un ejemplo:

```
DECLARE
  resultado temp.col1%TYPE;
  CURSOR c1 IS
    SELECT n1, n2, n3 FROM tabla_datos
      WHERE num_exper=1;
BEGIN
  FOR c1_rec IN c1 LOOP
    /* Calculamos y almacenamos los resultados */
    resultado:=c1_rec.n2 / (c1_rec.n1+c1_rec.n3);
    INSERT INTO temp VALUES (resultado, NULL, NULL);
  END LOOP;
  COMMIT;
END;
```

Cada Iteración del bucle, lleva implícito un fetch de la consulta asociada a un cursor. Al entrar en el bucle se abre el cursor, y al salir se cierra.

La manera de referenciar a los elementos del cursor, es mediante el nombre del identificador que hemos asociado al bucle FOR, y después un punto y el nombre del campo de la consulta que queramos referenciar.

3.4.1 Utilización de Subconsultas

No es necesario declarar un cursor si no se desea, puesto que PL/SQL **permite sustituir la referencia a un cursor por una subconsulta**. Veamos un ejemplo en el cual un bucle FOR calcula un bonus, e inserta los resultados en una tabla de Bbdd:

```
DECLARE
  bonus REAL;
BEGIN
  FOR emp_rec IN (SELECT num_emp, sal_emp, com_emp FROM
emp) LOOP
    bonus:=(emp_rec.sal_emp*0.05) + (emp_rec.com_emp*0.25);
    INSERT INTO bonus_emp VALUES (emp_rec.num_emp, bonus);
  END LOOP;
  COMMIT;
END;
```

De todas formas, y aunque esto en ocasiones puede ser práctico para evitarnos definir el cursor, se debe tener mucho cuidado al utilizarlo, puesto que puede dificultar mucho la lectura del código fuente, e incluso su mantenimiento posterior.

INNOV@
Desarrollos Informáticos

4 UNIDAD 7: MANEJO DE ERRORES

Objetivo general de la unidad

Explicar como PL/SQL resuelve los temas relacionados con el manejo de errores y excepciones.

Objetivos específicos

Que el alumno conozca las excepciones predefinidas por PL/SQL, y sepa a su vez, crear excepciones nuevas para el tratamiento de errores en sus programas.

Contenidos

Introducción

Ventajas de la Excepciones

Excepciones Predefinidas

Excepciones definidas por el usuario

INNOV@
Desarrollos Informáticos

Cuaderno de notas

4.1 Introducción

En PL/SQL, a un *warning* o una condición de error, se le llama ***excepción***. Las excepciones pueden ser Internas (definidas por el propio sistema), o definidas por el usuario. Ejemplos de excepciones definidas de forma interna son *division by zero* y *out of memory*. Las excepciones internas más comunes tienen nombres predefinidos, como por ejemplo ZERO_DIVIDE y STORAGE ERROR. De todas formas a las excepciones internas que no lo tienen, se les puede dar uno sin problemas.

El usuario puede definir excepciones propias en la parte declarativa del cualquier bloque, subprograma o package PL/SQL. Por ejemplo, podríamos definir una excepción llamada *fondos_insuficientes* para controlar las transacciones de un banco. Al contrario que el caso de las excepciones internas, a las excepciones definidas por el usuario **se les debe dar un nombre**.

Cuando ocurre un error, la excepción se dispara (*is raised*), es decir, la ejecución normal se detiene, y se transfiere el control a la parte de tratamiento de errores del bloque PL/SQL o subprograma. Las excepciones internas **son disparadas de forma implícita (automática) por el propio sistema**, sin embargo las excepciones definidas por el usuario **deben ser disparadas de forma explícita mediante sentencias RAISE**, las cuales pueden disparar también excepciones predefinidas.

Para el manejo de excepciones que se disparan, se deben escribir rutinas separadas que tienen el nombre de *exception handlers*. Cuando una rutina de este tipo de ejecuta, se detiene el funcionamiento del bloque que la ha disparado, y posteriormente el control pasa al bloque de nivel superior… en caso de no existir, el control pasaría al sistema (es decir, terminaría la ejecución del programa PL/SQL).

Veamos un ejemplo ahora, en el cual se calcula y almacena el ratio de los precios para que existan ganancias. Si la compañía tiene cero ganancias, se disparará la excepción ZERO_DIVIDE; esto dentendrá la ejecución normal del bloque, y transferirá el control a las rutinas de manejo de errores. El *handler* opcional OTHERS, agrupará todas las excepciones que el bloque no llama específicamente.

```
DECLARE
  pe_ratio NUMBER(3,1);
BEGIN
  SELECT precio/ganancias INTO pe_ratio FROM stocks
    WHERE simbolo='XYZ';  -- Esto puede causar el error de división entre cero…
  INSERT INTO estadis (simbolo,ratio) VALUES ('XYZ',pe_ratio);
  COMMIT;
EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- Controla la excepción…
    INSERT INTO estadis (simbolo,ratio) VALUES ('XYZ',NULL);
    COMMIT;
  …
  WHEN OTHERS THEN  -- Controla el resto de excepciones…
    ROLLBACK;
END;  -- El manejador de errores y el bloque acaban aquí…
```

Este último ejemplo ilustra el manejo de errores, no la óptima utilización del INSERT, puesto que nos podríamos haber ahorrado la excepción simplemente escribiendo la sentencia de la siguiente forma:

```
INSERT INTO estadis (simbolo,ratio)
  SELECT simbolo, DECODE(ganancias,0,NULL,precio/ganancias)
    FROM stocks WHERE simbolo='XYZ';
```

4.2 Ventajas de las excepciones

La utilización de las excepciones para el control de errores tiene muchas ventajas. Sin utilizarla, cada vez que ejecutamos ciertos comandos, debemos comprobar los errores… Veamos un ejemplo práctico de esto:

```
BEGIN
  SELECT …
    -- Controlamos el error 'no data found'…
  SELECT …
    -- Controlamos el error 'no data found'…
```

```
SELECT …
  -- Controlamos el error 'no data found'…
```

El procesamiento de errores no está claramente diferenciado del proceso normal… no es una programación robusta.

Veamos como deberían escribirse estas líneas de código…

```
BEGIN
  SELECT …
  SELECT …
  SELECT …
  …
EXCEPTION
  WHEN NO_DATA_FOUND THEN  -- Controla el error…
  …
END;
```

Las excepciones mejoran la lectura del código, ya que las rutinas principales no se ven mezcladas con las rutinas de manejo de errores. Además el hecho de utilizar excepciones nos asegura que si se dispara alguna excepción de ese tipo, nuestro programa en PL/SQL la tratará.

4.3 Excepciones Predefinidas

Una excepción interna se dispara cuando un programa PL/SQL viola una regla de Oracle, o excede un límite dependiente del sistema. Cada error de Oracle tiene un número, pero las excepciones pueden tener además un nombre. Por lo tanto, PL/SQL tiene predefinidos algunos errores de Oracle como excepciones. Por ejemplo, PL/SQL dispara la excepción predefinida NO_DATA_FOUND, si una sentencia SELECT no devuelve ninguna columna.

Para controlar otros errores de Oracle, se puede usar el manejador OTHERS. Las funciones de información de errores SQLCODE y SQLERRM, son especialmente útiles con el manejador OTHERS, puesto que devuelven el número y texto asociados a cualquier error que trate Oracle. Se puede utilizar el pragma EXCEPTION_INIT, para asociar nombres de excepciones a números de error de Oracle.

PL/SQL declara excepciones predefinidas de forma global en el package STANDARD, el cual está asociado al entorno PL/SQL. Veamos la tabla completa de errores.

Excepción	Error de Oracle	Valor SQLCODE
ACCESS_INTO_NULL	ORA-06530	-6530
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Veamos ahora una breve descripción para las excepciones predefinidas:

Excepción	Disparada cuando…
ACCESS_INTO_NULL	Se intenta asignar valores a los atributos de un Objeto no inicializado.
COLLECTION_IS_NULL	Se intenta aplicar métodos predefinidos (exceptuando EXISTS), sobre colecciones de Objetos como varrays o tablas de memoria no inicializados; o también cuando se intenta asignar valores a los elementos de una colección de Objetos que no ha sido inicializada.
CURSOR_ALREADY_OPEN	Se intenta abrir un cursor que ya está abierto.
DUP_VAL_ON_INDEX	Se intenta almacenar valores duplicados en una columna de Bbdd que tiene una constraint tipo Unique sobre esa columna.

INVALID_CURSOR	Se intenta realizar una operación no válida sobre un cursor, como por ejemplo cerrar un cursor que no ha sido abierto.
INVALID_NUMBER	En una sentencia SQL, la conversión de un string de caracteres a un número falla, porque el string no representa un número válido. En sentencias aisladas dentro de un procedimiento PL/SQL, se dispara la excepción VALUE_ERROR para este mismo caso.
LOGIN_DENIED	Se intenta entrar en Oracle con un username o password no válidos.
NO_DATA_FOUND	Una sentencia SELECT INTO no devuelve valores, o bien se referencia a un elemento borrado en una tabla de memoria, o bien se referencia a un elemento no inicializado en una tabla de memoria.
NOT_LOGGED_ON	El programa PL/SQL realiza una llamada a Bbdd sin estar conectado a Oracle.
PROGRAM_ERROR	PL/SQL tiene un problema interno.
ROWTYPE_MISMATCH	La variable del cursor del host, y la variable del cursor de PL/SQL, implicados en una sentencia de asignación, tienen tipos incompatibles.
STORAGE_ERROR	PL/SQL se queda sin memoria, o la memoria está estropeada.
SUBSCRIPT_BEYOND_COUNT	Se referencia a un elemento de un varray o tabla de memoria, utilizando un índice mayor que el número más largo que tiene como índice el varray o tabla de memoria.
SUBSCRIPT_OUTSIDE_LIMIT	Se referencia a un elemento de un varray o tabla de memoria, utilizando un índice que está fuera del rango legal (Por ejemplo –1)
TIMEOUT_ON_RESOURCE	Se produce un time-out cuando Oracle está esperando por un recurso.
TOO_MANY_ROWS	Una sentencia SELECT INTO devuelve más de una fila.
VALUE_ERROR	Se produce algún error de tipo aritmético, conversión, truncate, o constraint de tipo size.
ZERO_DIVIDE	Se intenta dividir entre cero.

4.4 Excepciones definidas por el usuario

PL/SQL permite la definición de excepciones por el usuario. Al contrario que la excepciones predefinidas, las excepciones definidas por el usuario deben ser declaradas y disparadas de forma explícita mediante sentencias RAISE.

4.4.1 Declaración de excepciones

Las excepciones se pueden declarar tan sólo en la parte declarativa de un bloque, subprograma o package PL/SQL. Se declara la excepción **introduciendo su nombre, y seguidamente la palabra clave EXCEPTION**. Veamos un ejemplo:

```
DECLARE
  mi_excepcion EXCEPTION;
```

Las declaraciones de variables y excepciones son muy parecidas, pero se debe recordar que **una excepción es una condición de error**, y no un elemento destinado a contener información. Al contrario que las variables, las excepciones no pueden aparecer en sentencias de asignación o sentencias SQL. Sin embargo, se aplican las mismas reglas de ámbito y visibilidad que en el caso de las variables.

4.4.2 Reglas de ámbito y visibilidad

No se puede declarar la misma excepción dos veces en el mismo bloque, sin embargo sí que se puede declarar la misma excepción en dos bloques distintos.

Las excepciones declaradas en un bloque se consideran locales a ese bloque, y globales para todos sus subbloques. Debido a que un bloque puede referenciar tan solo a excepciones locales o globales, los bloques incluidos en otros no pueden referenciar a excepciones declaradas en un subbloque.

Si se vuelve a declarar una excepción global en un subbloque, **la declaración local prevalecerá sobre la global**, por lo tanto el subbloque no podrá referenciar a la excepción global a menos que sea declarado en un bloque etiquetado… en ese caso la siguiente sintaxis será válida:

```
Etiqueta_bloque.nombre_excepcion
```

Vemos un ejemplo que ilustra las reglas de ámbito y visibilidad.

```
DECLARE
  excepcion1 EXCEPTION;
  num_acct NUMBER;
BEGIN
  …
  DECLARE  -- Empieza el subbloque
```

```
excepcion1 EXCEPTION;  -- Esta declaración prevalece…
num_acct NUMBER;
BEGIN
…
IF … THEN
  RAISE excepcion1;  -- Si se dispara no se tratará…
END IF;
…
END;  -- Fin del subbloque…
EXCEPTION
WHEN excepcion1 THEN  -- No tratará a la excepción interior…
…
END;
```

La excepción interior al subbloque que se dispara nunca se tratará, puesto que la definición local prevalece sobre la global. Debemos tener cuidado en ese aspecto.

4.4.3 Utilización de EXCEPTION_INIT

Para controlar excepciones internas que no tienen nombre, se debe usar el manejador OTHERS, o el *pragma* EXCEPTION_INIT. Un *pragma* es una **directiva de compilación**, la cual puede ser interpretada como una marca para el compilador. Los pragmas (también llamados *pseudoinstrucciones*), se procesan en tiempo de compilación, no en tiempo de ejecución. Por ejemplo, en el lenguaje Ada, el siguiente pragma le dirá al compilador que optimice el uso de espacio:

```
pragma OPTIMIZE(SPACE);
```

En PL/SQL, el pragma EXCEPTION_INIT le dice al compilador que **asocie un nombre de excepción con un número de error de Oracle**. Esto permitirá referenciar a cualquier excepción interna por el nombre, y escribir un código adecuado para su tratamiento.

Se incluye el pragma EXCEPTION_INIT en la parte declarativa de un bloque, subprograma, o package PL/SQL, utilizando la siguiente sintaxis:

```
PRAGMA EXCEPTION_INIT (nombre_excepcion,
Número_de_error_de_Oracle);
```

Donde nombre_excepcion es el nombre de una excepción previamente declarada. El pragma debe aparecer en algún lugar después de la declaración de la excepción y siempre en la misma parte declarativa, como se muestra en el siguiente ejemplo:

```
DECLARE
  deadlock_detectado EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detectado, -60);
BEGIN
  …
EXCEPTION
  WHEN deadlock_detectado THEN
    -- Tratamiento del Error…
  …
END;
```

4.4.4 Uso del RAISE_APPLICATION_ERROR

El package DBMS_STANDARD, el cual se proporciona con Oracle, proporciona algunas facilidades para ayudar a nuestras aplicaciones a interactuar con Oracle. Por ejemplo, el procedimiento RAISE_APPLICATION_ERROR nos permite dar mensajes de error definidos por el usuario en procedimientos almacenados. De esta forma, podemos informar de errores en la aplicación y evitar la devolución de excepciones indefinidas.

Para llamar al procedimiento RAISE_APPLICATION_ERROR, se utiliza la siguiente sintaxis:

```
RAISE_APPLICATION_ERROR(numero_de_error, mensaje[, {TRUE | FALSE}]);
```

Donde numero_de_error es un entero negativo comprendido entre el siguiente rango: –20000 … -20999, y mensaje es un string de caracteres de hasta 2048 bytes de longitud. Si el tercer parámetro opcional es TRUE, el error se almacena en la pila de errores previos, sin embargo cuando este parámetro es FALSE (es el valor por defecto), el error sustituye a todos los errores previos en la pila. El package DBMS_STANDARD es una extensión del package STANDARD, por lo tanto no tenemos que referenciarlo al llamar a una función de ese package.

Una aplicación puede llamar a RAISE_APPLICATION_ERROR solo desde un procedimiento almacenado; cuando es llamado, se finaliza la ejecución del procedimiento, y devuelve a la aplicación un número de error y mensaje definidos por el usuario. El número de error y el mensaje puede ser capturado como un error de Oracle.

Veamos un ejemplo en el que llamaremos a RAISE_APPLICATION_ERROR si el salario de un empleado se pierde:

```
CREATE PROCEDURE trata_salario (id_emp NUMBER,
incremento NUMBER) AS
  salario_actual NUMBER;
BEGIN
  SELECT emp_sal INTO salario_actual FROM emp
    WHERE num_emp=id_emp;
  IF salario_actual IS NULL THEN
    /* Damos el mensaje definido por el usuario */
    RAISE_APPLICATION_ERROR(-20101, 'Falta Salario');
  ELSE
    UPDATE emp SET emp_sal=salario_actual+incremento
      WHERE num_emp=id_emp;
    COMMIT;
  END IF;
END trata_salario;
```

La llamada a este procedimiento provocará una excepción de PL/SQL, la cual se puede procesar utilizando las funciones de información de errores SQLCODE y SQLERRM en un manejador OTHERS. También se puede utilizar el pragma EXCEPTION_INIT para asociar los números de error específicos que devuelve el procedimiento a excepciones propias… Veamos un ejemplo:

```
DECLARE
  …
  salario_nulo EXCEPTION;
  /* Asociamos el número de error que devuelve el procedimiento
      a la excepción definida arriba */
  PRAGMA EXCEPTION_INIT(salario_nulo, -20101);
BEGIN
```

```
  …
  trata_salario(numero_empleado, incremento);
EXCEPTION
  WHEN salario_nulo THEN
    INSERT INTO auditoria_emp VALUES (numero_empleado,…);
    COMMIT;
  …
END;
```

5

UNIDAD 8: SUBPROGRAMAS Y PACKAGES

Objetivo general de la unidad

Explicar como trata PL/SQL los conceptos de Subprogramas y Packages.

Objetivos específicos

Que el alumno sepa definir Procedimientos y Funciones, así como agruparlos posteriormente en Packages para mejorar la eficiencia y modularidad del aplicativo.

Contenidos

Ventajas de los Subprogramas

Procedimientos y Funciones

Recursividad en PL/SQL

Concepto de Package

Ventajas de los Packages

INNOV@
Desarrollos Informáticos

Cuaderno de notas

5.1 Ventajas de los subprogramas

Los subprogramas en PL/SQL, son bloques de PL/SQL con un determinado nombre, que pueden tener parámetros asociados, y ser llamados en cualquier momento. PL/SQL tiene dos tipos de subprogramas llamados *procedimientos (procedures)*, y *funciones (functions)*. Normalmente se utiliza un procedimiento para ejecutar una acción, y una función para calcular un valor.

Al igual que los bloques de PL/SQL sin nombre o *anónimos*, los subprogramas tienen una parte declarativa, en la cual se declaran las variables, tipos, cursores, constantes, excepciones, etc… una parte ejecutable, que contiene el código, y una parte opcional de tratamiento de excepciones.

5.1.1 Lista de las ventajas

Los subprogramas proporcionan *extensibilidad*, es decir, nos permiten aprovechar el lenguaje PL/SQL para cubrir todas las necesidades. Por ejemplo, si necesitamos un procedimiento que cree nuevos departamentos, podemos escribirlo de manera muy sencilla:

PROCEDURE crear_departamento (nuevo_nombre CHAR, nueva_dir CHAR) IS

BEGIN

INSERT INTO dep VALUES (seq_dept.NEXTVAL, nuevo_nombre, nueva_dir);

END crear_departamento;

Los subprogramas también proporcionan *modularidad*, es decir permiten dividir el código en unidades lógicas bien estructuradas.

Los subprogramas facilitan la *reutilización* de código y el *mantenimiento* de las aplicaciones. Una vez validado, un subprograma puede ser utilizado por muchas aplicaciones diferentes… es más, el subprograma solo cambia si se modifica su definición, lo cual simplifica el mantenimiento.

Por último, los subprogramas proporcionan *abstracción*, ya que se debe realizar la división en unidades lógicas, lo cual implica pensar **qué** es lo que debe hacer un subprograma (área del programa), y no solo **como** debe hacerlo.

5.2 Procedimientos y Funciones

Un procedimiento es un subprograma que implementa una determinada acción. La sintaxis general de un procedimiento es la siguiente:

PROCEDURE nombre [(parametro[, parametro, …])] IS

[declaraciones_locales]

BEGIN

sentencias_ejecutables

[EXCEPTION

manejadores de excepciones]

END [nombre];

Donde parametro sigue la siguiente sintaxis:

nombre_parametro [IN | OUT | IN OUT] tipo_de_dato [{:= | DEFAULT} expresion]

No se le puede poner la constraint de NOT NULL a un parámetro.

5.2.1 Funciones

Una función es un subprograma que calcula un valor. Las funciones y los procedimientos se estructuran igual, excepto que las funciones deben tener una claúsula RETURN, es decir, deben devolver un valor. La sintaxis general de una función es la siguiente:

FUNCTION nombre [(parametro[, parametro, …])] RETURN tipo_de_dato IS

[declaraciones_locales]

BEGIN

sentencias_ejecutables

[EXCEPTION

manejadores de excepciones]

END [nombre];

Donde parametro sigue la siguiente sintaxis:

nombre_parametro [IN | OUT | IN OUT] tipo_de_dato [{:= | DEFAULT} expresion]

Al igual que en el caso de los procedimientos, no se le puede poner la constraint de NOT NULL a un parámetro.

5.3 Recursividad en PL/SQL

Un programa recursivo **es aquel que se llama a sí mismo**. Hay que pensar en las llamadas recursivas como llamadas a otros programas que realizan la misma tareas, pero en ejecuciones diferentes, y con parámetros distintos. Cada llamada recursiva crea tantas nuevas instancias de elementos, como elementos haya declarados en el programa.

Hay que tener cuidado en donde ponemos las llamadas recursivas, puesto que si las ponemos en un bucle FOR, o entre sentencias OPEN y CLOSE, se abre un nuevo cursor para cada llamada, con la consiguiente degradación del sistema.

Hay que especificar siempre y de manera directa **cual es la condición de salida de un programa recursivo**, puesto que si no se hace generará un bucle infinito que acabará con una extinción de la memoria disponible, y por consiguiente se disparará la excepción STORAGE_ERROR.

Veamos un ejemplo sencillo de recursividad implementada en PL/SQL; resolveremos el problema por excelencia asociado a la recursividad, es decir, la obtención del factorial de un número.

```
FUNCTION fac (n POSITIVE) RETURN INTEGER IS
BEGIN
  IF n=1 THEN  -- Condición de salida…
    RETURN 1;
  ELSE
    RETURN n*fac(N-1);  -- Llamada recursiva…
  END IF;
END fac;
```

5.4 Concepto de Package y definición

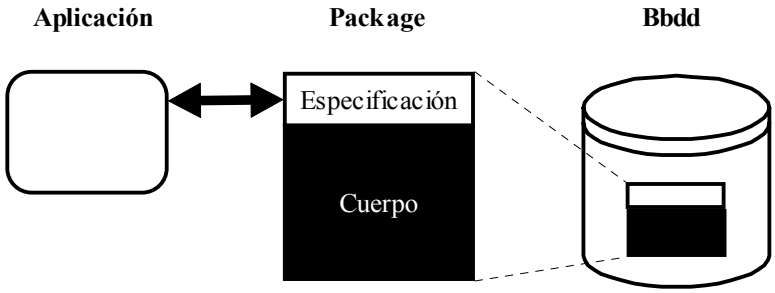
Un *package* es un objeto del esquema que agrupa de manera lógica tipos, elementos y subprogramas PL/SQL. Los packages tienen dos partes: una *especificación* y un *cuerpo*. La *especificación* es el interface para nuestras aplicaciones; allí se declaran los tipos, variables, constantes, excepciones, cursores y subprogramas que se pueden utilizar. El *cuerpo* define de forma completa los cursores y subprogramas… por lo tanto implementa la especificación.

Al contrario que los subprogramas, los packages **no pueden ser llamados ni parametrizados**. En cualquier caso, el formato de un package es parecido al de un subprograma:

```
CREATE PACKAGE nombre AS  -- Especificación (parte visible)
  -- Tipos públicos y declaración de Objetos…
  -- Especificación de subprogramas…
END [nombre];

CREATE PACKAGE BODY nombre AS  -- Cuerpo (parte oculta)
  -- Tipos privados y declaración de Objetos…
  -- Cuerpos de los subprogramas…
[BEGIN
  -- Sentencias de inicialización…]
END [nombre];
```

En la especificación encontramos declaraciones públicas, las cuales son visibles para nuestra aplicación, mientras que el cuerpo tiene la implementación y declaraciones privadas, que no son visibles para nuestra aplicación. Veamos un esquema gráfico:



5.4.1 Definir un package

Como ya hemos comentado, los packages tienen una especificación en la que se define la parte pública, y un cuerpo en la que se realiza la implementación de la parte pública, y si se desea, se añade alguna parte privada. La estructura está bastante clara, así que veamos un ejemplo de definición de package con una función y un procedimiento públicos, y dentro del cuerpo pondremos uno privado.

```
PACKAGE mi_package IS
  -- Declaraciones públicas…
  …
  PROCEDURE proc1(…);
  FUNCTION fun1(…) RETURN tipo_de_dato;
END mi_package;

PACKAGE BODY mi_package IS
BEGIN
  PROCEDURE proc2(…) IS  -- Procedimiento privado…
  BEGIN
    …
  END proc2;
  PROCEDURE proc1(…) IS
  BEGIN
    …
    proc2(…);  -- Este será privado…
  END proc1;
  FUNCTION fun1(…) RETURN tipo_de_dato IS
  BEGIN
    …
    RETURN …
  END fun1;
END mi_package;
```

5.5 Ventajas de los Packages

La utilización de packages tiene numerosas ventajas: modularidad, un diseño de las aplicaciones más sencillo, oculta información, añade funcionalidad, y proporciona un mejor rendimiento.

5.5.1 Modularidad

Los packages permiten encapsular de manera lógica tipos, objetos y subprogramas relacionados, en modulos PL/SQL con un mismo nombre. Cada package es sencillo de entender, y los interfaces entre los packages son simples, transparentes y bien definidos. Esto mejora el desarrollo de la aplicación.

5.5.2 Diseño de aplicaciones más sencillo

Cuando se diseña una aplicación, se debe inicializar toda la información del interface en la especificación… sin embargo no es necesario implementar todo el cuerpo, se puede ir haciendo poco a poco. Esto facilita mucho las cosas a la hora de realizar el diseño de aplicaciones.

5.5.3 Oculta información

Con los packages se puede especificar que tipos, objetos y subprogramas son públicos (visibles y accesibles) o privados. Por ejemplo, en un cuerpo de package podemos tener procedimientos que hayan sido declarados en la especificación, lo cual implica que son públicos, y otros que no… que son para uso interno, y por lo tanto no accesibles desde el exterior.

5.5.4 Añade funcionalidad

Las variables y cursores públicos persisten durante la duración de una sesión, por lo tanto pueden ser accesibles por todos los subprogramas que se ejecuten en el entorno. También permiten mantener datos a través de las transacciones, sin tener que almacenarla en la Bbdd.

5.5.5 Mejor rendimiento

Cuando se llama a un subprograma de un package, se almacena en memoria todo el package, por lo tanto cualquier llamada a un subprograma de dicho package que se efectúe en un espacio de tiempo relativamente corto, se ejecutará mucho más rápido.

6 ANEXO 3:EJERCICIOS

6.1 Ejercicios de la Unidad 4

1.- Determinar cuales de los siguiente Identificadores son equivalentes en PL/SQL

```
DECLARE
  Identificador1 INTEGER;
  Identificador_1 INTEGER;
  identificador1 INTEGER;
  IdEntificador_1 INTEGER;
  IDENTIFICADOR1 INTEGER;
```

2.- Determinar cual de los siguientes identificadores en PL/SQL es el válido:

```
DECLARE
  Primera variable VARCHAR2(40);
  end BOOLEAN;
  Una_variable VARCHAR2(40);
  Otra-variable VARCHAR2(40);
```

3.- ¿ Funcionaría la siguiente sentencia en PL/SQL?. En caso negativo proponer como resolverlo.

```
DECLARE
  nombre VARCHAR2(80);
  direccion VARCHAR2(80);
  tipo empleado VARCHAR2(4);
BEGIN
SELECT name, address, type INTO nombre, direccion, tipo empleado
  FROM emp;
```

```
END;
```

4.- ¿Qué resultado nos daría la siguiente comparación?

```
DECLARE
  identificador1 VARCHAR2(10):='Hola Pepe';
  identificador2 VARCHAR2(10):='Hola pepe';
BEGIN
  IF identificador1<>identificador2 THEN
    RETURN TRUE;
  ELSE
    RETURN FALSE;
  END IF;
END;
```

5.- Indicar que errores existen en el siguiente código fuente:

```
DECLARE
  a PLS_INTEGER:=1;
  b PLS_INTEGER:=6;
  salida_bucle BOOLEAN;
BEGIN
  salida_bucle:='FALSE';
  WHILE NOT salida_bucle LOOP
  BEGIN
    IF a>=b THEN
      salida_bucle:='TRUE';
    ELSE
      a:=(a+1);
    END IF;
  END LOOP;
END;
```

6.- En las siguientes sentencias de definición de Subtipos, hay una que no es válida... Indicar cual es, y como solucionarlo.

```
DECLARE
  SUBTYPE numero1 IS NUMBER;
  SUBTYPE cadena IS VARCHAR2(10);
  total numero1(6,2);
```

7.- ¿Qué resultado dará la ejecución del siguiente código?

```
DECLARE
  temp NUMBER(1,0);
  SUBTYPE numero IS temp%TYPE;
  valor numero;
BEGIN
  WHILE valor<20 LOOP
    valor:=valor+1;
  END LOOP;
END;
```

8.- En la siguiente frase, hay una serie de afirmaciones ciertas y otras que no lo son. Indicar cuales son ciertas y cuales falsas, y para éstas últimas razonar el por qué.

** PL/SQL es capaz de realizar conversiones entre tipos de datos, de hecho puede convertir cualquier tipo de dato en otro diferente. Existen dos tipos de conversiones en PL/SQL: las conversiones Implícitas y las conversiones Explícitas; por ejemplo la función predefinida TO_DATE realiza una conversión implícita de un tipo STRING a un tipo DATE. Las conversiones son muy eficientes (sobre todo las implícitas), y debemos aprovechar al máximo esta funcionalidad soportada por PL/SQL.*

9.- Indicar del siguiente juego de declaraciones, cuales son correctas y cuales no, indicando además por qué no son correctas.

```
DECLARE
  cierto BOOLEAN:=FALSE;
```

id_externo INTEGER(4) NOT NULL;

cosa PLS_INTEGER:=2;

*producto PLS_INTEGER:=2*cosa;*

suma PLS_INTEGER:=cosa + la_otra;

la_otra PLS_INTEGER:=2;

tipo_uno NUMBER(7,2) NOT NULL:=3;

tipo_otro tipo_uno%TYPE;

10.- Suponiendo que tenemos una tabla llamada EMP, en la cual existen tres campos: nombre VARCHAR2(40), direccion VARCHAR2(255), telefono INTEGER(10)... ¿Qué valores podrán tomar las variables que definimos en la siguiente declaración?

DECLARE

valor1 emp%ROWTYPE;

valor2 emp.nombre%TYPE;

valor3 emp.telefono%TYPE;

CURSOR c1 IS

SELECT nombre,direccion FROM emp;

valor4 c1%ROWTYPE;

valor5 emp%ROWTYPE;

11.- En base al enunciado anterior... ¿Sería correcta la siguiente asignación?

valor1:=valor5;

¿y esta?

valor4:=valor1;

Razonar el por qué en ambos casos.

12.-¿ Es esta declaración correcta en PL/SQL?

DECLARE

i, j INTEGER;

¿Y si la escribieramos así?

DECLARE

i, j INTEGER, INTEGER;

13.- Supongamos que tenemos un procedimiento llamado calculo_integral, el cual está incluido en un package llamado el_package_de_los_calculos, que su vez se encuentra en la instancia de Bbdd llamada ORCL. ¿Como se haría para llamar a dicho procedimiento en el caso de encontrarse conectado a la Bbdd ORCL con un usuario que tuviera acceso al procedimiento?, ¿Y en el caso de no estar en ORCL?

14.- ¿Funcionaria el siguiente código?. Explicar por qué y como solucionarlo (si se os ocurren varias formas de arreglarlo, explicarlas todas).

```
DECLARE
  base PLS_INTEGER:=100;
BEGIN
  FOR dept IN 1..10 LOOP
    UPDATE dept SET nom_dept=base+dept
      WHERE cod_dept=base+dept;
  END LOOP;
END;
```

15.- ¿Qué ocurriría al ejecutar el siguiente código?, ¿Por qué?, ¿Cómo arreglarlo? (Si existen varias posibilidades, comentarlas e indicar cual sería más eficiente).

```
DECLARE
  ap1_emp VARCHAR2(40):='Fernandez';
BEGIN
  DELETE FROM emp WHERE ap1_emp=ap1_emp;
  COMMIT;
END;
```

16.- Definir brevemente los conceptos de ámbito y visibilidad. Explicar como los interpreta PL/SQL para el caso de los objetos definidos, y cual es el método más habitual para solucionar los problemas de visibilidad para variables con el mismo nombre en bloques diferentes.

17.- ¿Qué valor contendrá la variable 'sumador' al salir del bucle?, ¿Por qué?

```
DECLARE
  sumador PLS_INTEGER;
BEGIN
  FOR i IN 1..100 LOOP
    sumador:=sumador+i;
  END LOOP;
END;
```

18.-¿ Funcionaría el siguiente trozo de código?, ¿Por qué?, ¿Cómo arreglarlo?

```
DECLARE
  mi_valor PLS_INTEGER;
  cierto BOOLEAN:=FALSE;
BEGIN
  WHILE NOT cierto LOOP
    IF mi_valor=NULL THEN
      mi_valor:=1;
    ELSE
      mi_valor:=mi_valor+1;
    END IF;
    IF mi_valor>100 THEN cierto:=TRUE; END IF;
    EXIT WHEN cierto;
  END LOOP;
END;
```

19.- En la utilización del Operador LIKE, ¿qué dos caracteres especiales tenemos?, ¿para que se utilizan?

6.2 Ejercicios de la Unidad 5

1.- ¿Es correcta la siguiente sintaxis General de la sentencia IF-THEN-ELSE?, ¿Por qué?, ¿Cómo la escribirías?.

```
BEGIN
  IF condicion1 THEN
  BEGIN
    secuencia_de_instrucciones1;
  ELSE
    secuencia_de_instrucciones2;
  ENDIF;
END;
```

2.- Escribir la sintaxis General de un código que evalúe si se cumple una condición, en caso de cumplirse que ejecute una serie de sentencias, en caso contrario que evalúe otra, que de cumplirse ejecute otras instrucciones, si ésta no se cumple que evalúe una tercera condición.. y así N veces. En caso de existir varias soluciones, comentarlas y escribir la más óptima o clara.

3.- Implementar en PL/SQL un bucle infinito que vaya sumando valores en una variable de tipo PLS_INTEGER.

4.- En base al bucle anterior, añadirle la condición de que salga cuando la variable sea mayor que 10.000.

5.- Implementar un bucle en PL/SQL mediante la sentencia WHILE, en el cual vayamos sumando valores a una variable mientras ésta sea menor que 10, y asegurándonos de que el bucle se ejecuta por lo menos una vez.

6.- Implementar en PL/SQL, el código necesario de un programa que al final de su ejecución haya almacenado en una variable llamada 'cadena', el siguiente valor:

*cadena:='10*9*8*7*6*5*4*3*2*1'*

No es necesario definirlo como un procedimiento o función, pero sí declarar todas las variables necesarias para su correcto funcionamiento.

7.- ¿Es correcto el siguiente código en PL/SQL?, ¿Por qué? Nota: Ignorar el funcionamiento del código (no hace nada), ceñirse exclusivamente a la sintaxis válida en PL/SQL.

```
FOR ctr IN 1..10 LOOP
  IF NOT fin THEN
    INSERT INTO temp
     VALUES (ctr, 'Hola');
    COMMIT;
    factor:=ctr*2;
  ELSE
    ctr:=10;
  END IF;
END LOOP;
```

8.- ¿Qué resultado provoca la ejecución del siguiente código PL/SQL?

```
DECLARE
  variable PLS_INTEGER:=1;
  almacenamos PLS_INTEGER:=1;
BEGIN
  FOR i IN 5..variable LOOP
    almacenamos:=almacenamos+i;
  END LOOP;
  INSERT INTO traza VALUES(TO_CHAR(almacenamos));
  COMMIT;
END;
```

9.- ¿Dónde tienen valor las variables de control de un bucle FOR?, ¿Hay alguna excepción?.

10.- ¿Qué ocurre cuando ejecutamos una sentencia NULL?, ¿Qué utilidad tiene?

11.- ¿Cuándo es especialmente útil la sentencia GOTO?, ¿Cuándo debemos usarla?

12.- ¿Qué resultado provocaría la ejecución del siguiente código?

```
BEGIN
  IF 2>3 THEN
    INSERT INTO TRAZA VALUES('Pongo un valor');
  ELSE
    NULL;
    INSERT INTO TRAZA VALUES('Pongo otro valor');
  END IF;
  COMMIT;
END;
```

6.3 Ejercicios de la Unidad 6

1.- ¿Es correcta la siguiente asignación en PL/SQL?, ¿Por qué?

```
DECLARE
  numero1 PLS_INTEGER;
  numero2 PLS_INTEGER;
BEGIN
  ...
  numero1:=MAX(numero2);
  ...
END;
```

2.- ¿Qué pseudocolumna de SQL nos proporciona el identificador físico de una fila en una tabla de la Bbdd?. ¿Y el número de columna

asociado a una consulta?. ¿Qué pseudocolumnas nos facilitan trabajar con secuencias?

3.- ¿Qué tipo de binding para las variables utiliza PL/SQL?, ¿Qué limitaciones tiene?, ¿Cómo las soluciona?.

4.- ¿Qué dos métodos generales tenemos para trabajar con Cursores en PL/SQL?, ¿Cual es más cómodo?

5.- Suponer que tenemos una tabla de empleados llamada 'emp' en la cual tenemos definidos entre otros los siguientes campos: emp_idemp, emp_nombre, emp_apellidos, emp_sexo, emp_colorpelo... donde emp_sexo será 'H' para los hombres y 'M' para las mujeres, y emp_colorpelo será un identificador del color de su pelo. Suponer que tenemos otra tabla llamada colorpelo_perfil_psico en la cual tenemos entre otras campos los siguientes cpp_colorpelo, cpp_nombrecolor, cpp_caractperf... el primero se refiere al identificador del color, el segundo a su nombre vulgar, y el tercero es una de las características psicológicas de ese perfil. Podemos tener N características para cada color de Pelo... sin embargo hay una que es preocupante y es la que pone 'Tendencias homicidas'. Bueno, bajo estos supuestos se pide la codificación de un algoritmo en PL/SQL que sea capaz de escribir en una tabla llamada TRAZA compuesta de un solo campo VARCHAR2(2000), a los sujetos que tengan esta característica... indicando además su nombre, apellidos y sexo.

Nota: Para simplificar, supondremos todos los campos de las tablas como tipo VARCHAR2(500).

6.- Explicar y definir los tipos de datos necesarios para realizar un programa en PL/SQL que recorra un cursor cualquiera, pero que deba en cada iteración referenciar a los valores que tenía el cursor en la anterior iteración.

6.4 Ejercicios de la Unidad 7

1.- ¿Qué diferencias principales existen entre las excepciones internas y las excepciones definidas por el usuario?

2.- ¿Cómo se declaran las excepciones que puede definir un usuario?, ¿Cómo se disparan?, ¿Qué reglas de ámbito y visibilidad siguen las excepciones?...

3.- ¿Pára que sirve el pragma EXCEPTION_INIT?, ¿Y qué es un pragma?

6.5 Ejercicios de la Unidad 8

1.- ¿Qué tipo de Subprogramas soporta PL/SQL?, ¿Qué diferencias existen entre ellos?

2.- Explicar el concepto de Package. ¿Qué ventajas ofrece?..

3.- Implementar de manera recursiva un subprograma en PL/SQL que calcule un número real 'x' elevado a un número entero 'n'.