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 ….
```

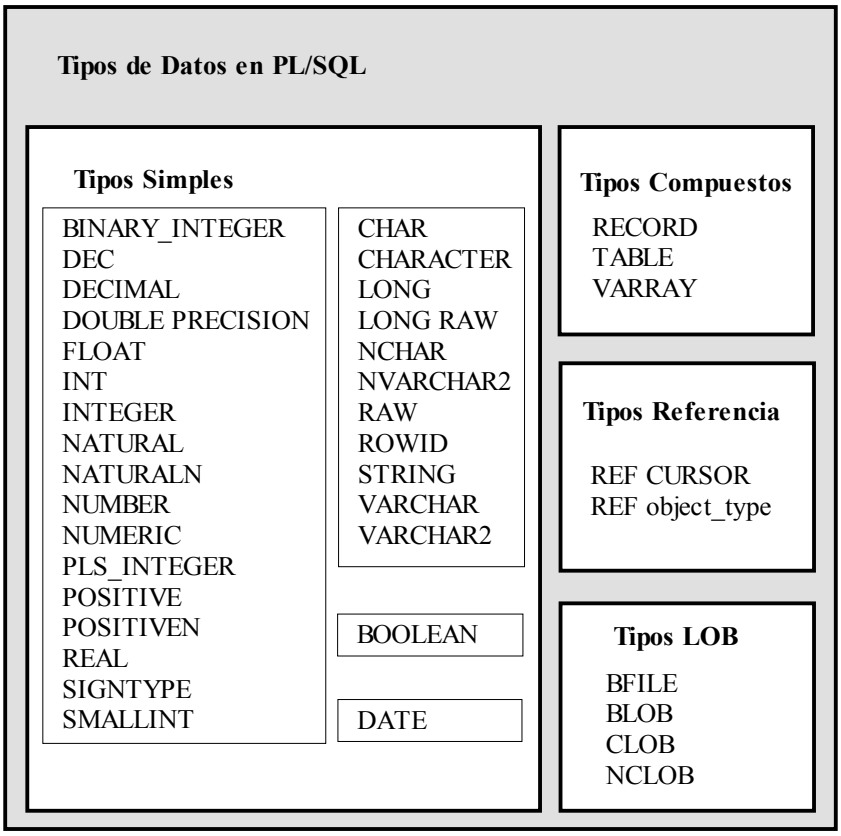
1.2 Tipos de Datos

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:

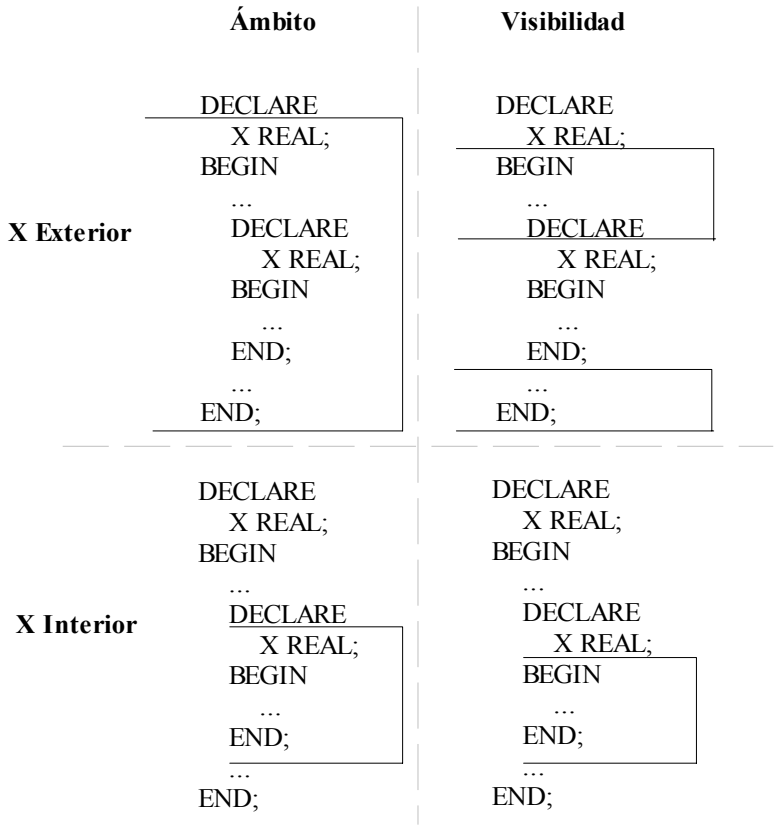
```
<<main>>
DECLARE
 nom_emp VARCHAR2(10) := 'Pedro';
BEGIN
 DELETE FROM emp WHERE nom_emp = main.nom_emp;
```

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

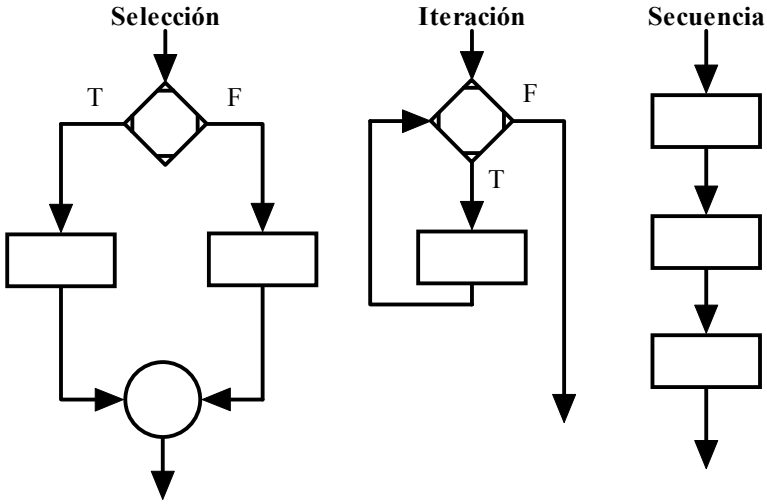
Control Secuencial

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.