

# Fundamentos del lenguaje PL/SQL

En esta unidad aprenderás a:

- 1 Declarar variables y otros objetos empleando los tipos de datos disponibles.**
- 2 Manejar operadores, funciones predefinidas y otros elementos del lenguaje.**
- 3 Controlar el flujo de ejecución de nuestros programas.**
- 4 Realizar procedimientos y funciones para desarrollar programas.**
- 5 Usar parámetros de distintos tipos.**



## 9.1 Introducción

En la unidad anterior hemos estudiado las características generales de PL/SQL y su utilización desde una perspectiva global y un tanto intuitiva. En esta unidad nos ocuparemos de los principales elementos del lenguaje PL/SQL: tipos de datos, variables, operadores y expresiones, reglas sintácticas, estructuras de control, así como de su uso en procedimientos y funciones.

## 9.2 Tipos de datos

PL/SQL dispone de los mismos tipos de datos que SQL, además de otros propios. En la mayoría de los casos existe compatibilidad con los tipos correspondientes soportados por la base de datos pero también existen diferencias que deben tenerse en cuenta.

Podemos clasificar los tipos de datos soportados por PL/SQL en las siguientes categorías:

- **Escalares:** almacenan valores simples. A su vez pueden subdividirse en:
  - *Carácter/Cadena:* CHAR, NCHAR, VARCHAR2, NVARCHAR2, LONG, RAW, LONG RAW, ROWID, UROWID.
  - *Numérico:* NUMBER, BINARY\_INTEGER, PLS\_INTEGER, BINARY\_DOUBLE, BINARY\_FLOAT.
  - *Booleano.*
  - *Fecha/hora:* DATE, TIMESTAMP, INTERVAL.
  - *Otros:* ROWID, UROWID.
- **Compuestos:** Son tipos compuestos de otros simples. Los veremos en la unidad de programación avanzada. Por ejemplo:
  - *Tablas indexadas.*
  - *Tablas anidadas.*
  - *Varrays.*
  - *Objetos.*
- **Referencias:** Difieren de los demás por sus características de manejo y almacenamiento.
  - *REF CURSOR:* Son referencias a cursosres.
  - *REF:* Son punteros a objetos.

En PL/SQL el programador puede definir sus propios tipos a partir de los anteriores.



## 9. Fundamentos del lenguaje PL/SQL

### 9.2 Tipos de datos

- **LOB:** Almacenan objetos de grandes dimensiones (*Large OBject*) de hasta 128 terabytes a partir de la versión 10gR1. Sustituyen con ventaja a los tipos LONG y LONG RAW.

A continuación, trataremos los datos escalares más utilizados, dejando para la de programación avanzada los datos compuestos, referencias y objetos.

#### A. Tipos escalares

En las siguientes tablas podemos observar los distintos tipos de datos escalares soportados por PL/SQL y sus descripciones.

##### CARÁCTER

Almacena **cadenas** de caracteres cuya longitud máxima es 32 KBytes. Al especificar las longitudes debemos tener en cuenta que, por defecto, cada carácter ocupa 1 byte, aunque hay juegos de caracteres que ocupan más (por ejemplo, el asiático).

##### CHAR[(L)]

Almacena cadenas de caracteres de **longitud fija**. Opcionalmente se puede especificar, entre paréntesis, la longitud máxima; en caso contrario, el valor por defecto es 1 y no se pone el paréntesis. Las posiciones no utilizadas **se rellenan con blancos**.

Ejemplo:

```
Nombre CHAR(15);  
Letra_nif CHAR;  
Situacion CHAR(); --ERROR
```

La longitud de **CHAR** se expresa por defecto en **bytes** pero se puede expresar en caracteres del juego de caracteres nacional utilizando la opción CHAR:

```
Apellido CHAR(25 CHAR);
```

La longitud en **NCHAR** se expresa siempre en caracteres del juego de **caracteres** nacional. Equivale a usar la opción CHAR comentada arriba:

```
Apellido NCHAR(25); -- equivalente a la anterior
```

Almacena cadenas de caracteres de **longitud variable** cuyo límite máximo se debe especificar.

Ejemplos:

```
Apellidos VARCHAR2(25);  
Control VARCHAR2; --ERR falta la longitud  
Nombre VARCHAR2(15 CHAR);  
Nombre NVARCHAR2(25);
```

También está disponible el tipo VARCHAR por compatibilidad con el estándar SQL, aunque Oracle desaconseja su utilización recomendando utilizar en su lugar VARCHAR2 y NVARCHAR2 para beneficiarse de la evolución de estos tipos.

##### LONG[(L)]

Almacena cadenas de longitud variable. Es similar a VARCHAR2 pero no permite la opción CHAR.

Ejemplos:

```
v_Observaciones LONG(5000);  
Resumen LONG;
```

Debemos tener en cuenta que el límite para este tipo en variables PL/SQL es de 32 KB, pero las columnas de este tipo admiten hasta 2GB.

## 9. Fundamentos del lenguaje PL/SQL

### 9.2 Tipos de datos



#### NUMÉRICOS

##### NUMBER(P, E)

Donde *P* es la precisión (número total de dígitos) y *E* es la escala (número de decimales).

La precisión y la escala son opcionales, pero si se especifica la escala hay que indicar la precisión.

Ejemplo:

```
Importe NUMBER(5,2); -- admite hasta 999.99  
Centimos NUMBER(,2); -- ERROR lo correcto es  
Centimos NUMBER(2,2); -- admite hasta .99
```

Si al asignar un valor se excede la escala, se producirá *truncamiento*. Si se excede la precisión, se producirá un *error*.

Ejemplo:

```
Importe:= 1000; -- ERROR excede la escala  
Importe:= 234.344; -- guarda 234.34
```

PL/SQL dispone de subtipos de NUMBER que se utilizan por compatibilidad y/o para establecer restricciones. Son DECIMAL, NUMERIC, INTEGER, REAL, SMALLINT, etcétera.

##### BINARY\_INTEGER

Es un tipo numérico entero que se almacena en memoria en formato binario para facilitar los cálculos. Puede contener valores comprendidos entre -2147483647 y + 2147483647.

Se utiliza en contadores, índices, etcétera.

Por ejemplo:

```
Contador BINARY_INTEGER;
```

Subtipos: NATURAL y POSITIVE.

##### PLS\_INTEGER

Similar a BINARY\_INTEGER y con el mismo rango de valores, pero tiene dos ventajas respecto al anterior:

- Es más rápido.
- Si se da desbordamiento en el cálculo se produce un error y se levanta la excepción correspondiente, lo cual no ocurre con BINARY\_INTEGER.

Por ejemplo:

```
indice PLS_INTEGER;
```

##### BINARY\_DOUBLE

Disponibles desde la versión 10gR1, su utilización se circunscribe al ámbito científico para realizar cálculos muy precisos y complejos conforme a la norma IEEE 754. Oracle no los soporta directamente como columnas en la base de datos. No los utilizaremos en este libro. Remitimos al lector a la documentación de Oracle para su eventual utilización.

##### BINARY\_FLOAT

#### BOOLEANO

##### BOOLEAN

Almacena valores lógicos TRUE, FALSE y NULL. Para representar estos valores **no debemos utilizar comillas**.

Ejemplo:

```
v_ocupado BOOLEAN;  
v_eliminado BOOLEAN DEFAULT FALSE;
```

La base de datos no soporta este tipo para definir columnas.

#### FECHA / HORA

Almacenan valores de tiempo. Son idénticos a los tipos correspondientes de la base de datos.

##### DATE

Almacena fechas incluyendo la hora. Los formatos en que muestran la información son los establecidos por defecto, aunque se pueden especificar máscaras de formato. Por defecto, sólo se muestra la fecha, pero también se almacena la hora: día/mes/año horas:minutos:segundos .

##### TIMESTAMP [(P)]

**TIMESTAMP** también almacena fecha y hora pero incluyendo fracciones de segundo. La precisión de estas fracciones se puede especificar como parámetro (por defecto es 6, es decir, 999999 millonésimas de segundo, pero puede variar entre 0 y 9). Normalmente el valor se toma de la variable del sistema SYSTIMESTAMP.

##### TIMESTAMP [(P)] WITH TIME ZONE

**TIMESTAMP WITH TIME ZONE** incluye el valor de la **zona horaria**.



## 9. Fundamentos del lenguaje PL/SQL

### 9.2 Tipos de datos

**TIMESTAMP [(P)]  
WITH LOCAL  
TIME ZONE**

**TIMESTAMP WITH LOCAL TIME ZONE** igual que **TIMESTAMP**, pero:

- Los datos **se almacenan normalizados a la zona horaria donde se encuentra la base de datos**.
- Los datos recuperados **se muestran en el formato de la zona horaria correspondiente a la sesión** de los usuarios.

El siguiente ejemplo ilustra estos conceptos:

```
DECLARE
    fechati TIMESTAMP;
    fechatz TIMESTAMP WITH TIME ZONE;
    fechatl TIMESTAMP WITH LOCAL TIME ZONE;
BEGIN /* incluiremos dos instrucciones en la misma linea */
    fechati := SYSTIMESTAMP;    DBMS_OUTPUT.PUT_LINE('fechati:'||fechati);
    fechatz := SYSTIMESTAMP;    DBMS_OUTPUT.PUT_LINE('fechatz:'||fechatz);
    fechatl := SYSTIMESTAMP;    DBMS_OUTPUT.PUT_LINE('fechatl:'||fechatl);
END;
```

El resultado de la ejecución será:

```
fechati:02/04/06 10:07:40,196000
fechatz:02/04/06 10:07:40,196000 +02:00
fechatl:02/04/06 10:07:40,196000
```

En este caso, FechaTL tiene el mismo valor que Fechati porque la zona horaria de almacenamiento y recuperación es la misma.

**INTERVAL YEAR  
[(PY)] TO MONTH**

Los subtipos de INTERVAL representan intervalos de tiempo entre dos fechas. La diferencia entre ellos es la manera de expresar el intervalo:

- INTERVAL YEAR TO MONTH se expresa en años/meses. Opcionalmente se puede incluir la precisión para el número de años, entre 0 y 9 dígitos, por defecto es 2.

```
v_anios_meses2 INTERVAL YEAR TO MONTH := '1-06';
```

- INTERVAL DAY TO SECOND se expresa en días/segundos. Opcionalmente se puede incluir la precisión para el número de días (entre 0 y 9 dígitos, por defecto es 2), y para el número de segundos (entre 0 y 9 dígitos, por defecto es 6).

```
v_dias_segundos INTERVAL DAY TO SECOND DEFAULT '02 14:0:0.0';
```

Estos tipos se utilizan cuando se requiere manejar diferencias de tiempo con precisiones expresadas en fracciones de segundo. En la mayoría de los casos recurriremos a los tipos tradicionales junto con las funciones disponibles MONTHS\_BETWEEN, etcétera.

### OTROS TIPOS ESCALARES

<b>RAW(L)</b>
Almacena datos binarios en longitud fija. Se utiliza para almacenar cadenas de caracteres evitando problemas por las conversiones entre conjuntos de caracteres que realiza Oracle.
<b>LONG RAW</b>
Almacena datos binarios en longitud variable evitando conversiones entre conjuntos de caracteres.

**Tabla 9.1.** Tipos de datos escalares en PL/SQL.

## 9. Fundamentos del lenguaje PL/SQL

9.3 Identificadores



### B. Principales diferencias de almacenamiento con la base de datos Oracle

Aunque PL/SQL ofrece soporte a los tipos de datos de las columnas de la base de datos Oracle, en algunos casos, no existe una equivalencia exacta entre los tipos de PL/SQL y los mismos tipos de las columnas de Oracle, especialmente en lo relativo a las longitudes máximas que pueden almacenar. La siguiente tabla revela las diferencias más significativas.

DIFERENCIAS DE ALMACENAMIENTO ENTRE PL/SQL Y LAS COLUMNAS DE LA BASE DE DATOS ORACLE

TIPO	VARIABLES PL/SQL	COLUMNAS DB ORACLE
VARCHAR2	32.767 bytes	4000 bytes
NVARCHAR2		
CHAR	32.767 bytes	4000 bytes
NCHAR		
LONG	32.767 bytes	2 GB
RAW	32.767 bytes	2.000 bytes
LONG RAW	32.767 bytes	2 GB
BOOLEAN	DISPONIBLE	NO SOPORTADO

Oracle realiza conversiones automáticas de tipos de datos (de manera acertada en muchos casos). No obstante, es preferible especificar las conversiones entre tipos empleando las funciones estudiadas en SQL.

### 9.3 Identificadores

Los **identificadores** se utilizan para nombrar los objetos que intervienen en un programa: variables, constantes, cursor, excepciones, procedimientos, funciones, etiquetas, etcétera.

En PL/SQL deben cumplir las siguientes características:

- Pueden tener entre 1 y 30 caracteres de longitud.
- El primer carácter debe ser una letra.
- Los restantes caracteres deben ser caracteres alfanuméricos o signos admitidos (letras, dígitos, los signos de dólar, almohadilla y subguion).
- No pueden incluir signos de puntuación, espacios, etcétera.

Se pueden saltar algunas de estas reglas utilizando identificadores entre comillas dobles (por ejemplo "2^varia ble/ ") pero no es aconsejable.

Ejemplos de identificadores válidos v\_ A#\$ X2 anio v\_num.

Ejemplos de identificadores no válidos \_v #A 2X año v-num.

PL/SQL no diferencia entre mayúsculas y minúsculas en los identificadores ni en las palabras reservadas. Por ejemplo, podemos escribir V\_Num\_Meses, v\_num\_meses o V\_NUM\_MESES. En los tres casos se trata del mismo identificador.



## 9. Fundamentos del lenguaje PL/SQL

### 9.4 Variables

## 9.4 Variables

Las **variables** sirven para almacenar información cuyo valor puede cambiar a lo largo de la ejecución del programa.

### A. Declaración e inicialización de variables

Todas las variables PL/SQL deben declararse en la *sección declarativa* antes de su uso. El formato genérico para declarar una variable es el siguiente:

```
<nombre_de_variable> <tipo> [NOT NULL] [{:= | DEFAULT}
                                         <valor>];
```

La opción **DEFAULT** (o bien la asignación «`:=`») sirve para asignar valores por defecto a la variable desde el momento de su creación.

```
DECLARE
    importe NUMBER(8,2);
    contador NUMBER(2,0) := 0;
    nombre CHAR(20) NOT NULL := "MIGUEL";
    . . .
```

Para cada variable se debe especificar el *tipo*. No se puede, como en otros lenguajes, indicar una lista de variables del mismo tipo y, a continuación, el tipo. PL/SQL no lo permite.

La opción **NOT NULL** fuerza a que la variable tenga siempre un valor. Si se usa, deberá inicializarse la variable en la declaración con **DEFAULT** o con `:=` para la inicialización. Por ejemplo:

```
nombre CHAR(20) NOT NULL DEFAULT "MIGUEL";
```

También se puede inicializar una variable que no tenga la opción **NOT NULL**. Por ejemplo:

```
nombre CHAR(20) DEFAULT "MIGUEL";
```

Si no se inicializan las variables en PL/SQL, se garantiza que su valor es **NULL**.



### Actividades propuestas

- 1 Indica los errores que aparecen en las siguientes instrucciones y la forma de corregirlos.

```
DECLARE
    Num1 NUMBER(8,2) := 0
    Num2 NUMBER(8,2) NOT NULL := 0;
```

(Continúa)



(Continuación)

```

Num3 NUMBER(8,2) NOT NULL;
Cantidad INTEGER(3);
Precio, Descuento NUMBER(6);
Num4 Num1%ROWTYPE;
Dto CONSTANT INTEGER;
BEGIN
  ...
END;
  
```

## B. Uso de los atributos %TYPE y %ROWTYPE

En lugar de indicar explícitamente el tipo y la longitud de una variable existe la posibilidad de utilizar los atributos **%TYPE** y **%ROWTYPE** para declarar variables que sean del mismo tipo que otros objetos ya definidos.

- **%TYPE** declara una variable del mismo tipo que otra, o que una columna de una tabla. Su formato es:

```
nombre_variable nombre_objeto%TYPE;
```

- **%ROWTYPE** declara una variable de registro cuyos campos se corresponden con las columnas de una tabla o vista de la base de datos. Su formato es:

```
nombre_variable nombre_objeto%ROWTYPE;
```

Ejemplos:

- `total importe%TYPE;`

Declara la variable `total` del mismo tipo que la variable `importe` que se habrá definido previamente.

- `nombre_moroso clientes.nombre%TYPE;`

Declara la variable `nombre_moroso` del mismo tipo que la columna `nombre` de la tabla `clientes`.

- `moroso clientes%ROWTYPE;`

Declara la variable `moroso` que podrá contener una fila de la tabla `clientes`.

Para hacer referencia a cada uno de los campos indicaremos el nombre de la variable, un punto y el nombre del campo que coincide con el de la columna correspondiente. Por ejemplo:

```
DBMS_OUTPUT.PUT_LINE(moroso.nombre);
```

Al declarar una variable del mismo tipo que otro objeto usando los atributos **%TYPE** y **%ROWTYPE**, se hereda el tipo y la longitud, pero no los posibles atributos `NOT NULL` ni los valores por defecto que tuviese definidos el objeto original.



## 9. Fundamentos del lenguaje PL/SQL

### 9.4 Variables

#### C. Ámbito y visibilidad de las variables

Las **variables** se crean al comienzo del bloque y dejan de existir una vez finalizada la ejecución del bloque en el que han sido declaradas. El ámbito de una variable incluye el bloque en el que se declara y sus bloques «hijos».

Una variable declarada en un bloque será **local** para el bloque en el que ha sido declarada y **global** para los bloques hijos de éste. Las variables declaradas en los bloques hijos no son accesibles desde el bloque padre.

En el siguiente ejemplo, podemos observar que `v1` es accesible para los dos bloques (es local al bloque padre y global para el bloque hijo), mientras que `v2` solamente es accesible para el bloque hijo.

```
DECLARE          -----Bloque PADRE
    v1 CHAR;
BEGIN
    v1 := 27;

DECLARE          -----Bloque HIJO
    v2 CHAR;
BEGIN
    v2:= 2;
    v1:= v2;
    ...
END;           -----Fin bloque HIJO

v2:= v1; --> Error: v2 no existe en este ámbito.

END;           -----Fin bloque PADRE
```

En el caso de que un identificador local coincida con uno global, si no se indica más, se referencia el local. Es decir, el identificador local dentro de su ámbito oculta la visibilidad del global. No obstante, se pueden utilizar etiquetas y cualificadores para deshacer ambigüedades.

```
<<padre>>   -- etiqueta que identifica al bloque padre.
DECLARE
    v CHAR;
BEGIN
    ...
    DECLARE
        v CHAR;
    BEGIN
        ...
        v := padre.v;-- el primero es el identificador local
        ...
    END;
END;
```

En caso de coincidencia, los identificadores de columnas tienen precedencia sobre las variables y parámetros formales; éstos, a su vez, tienen precedencia sobre los nombres de tablas.



## 9.5 Constantes y literales

Para representar valores que no variarán a lo largo de la ejecución del programa disponemos de dos elementos: *constantes* y *literales*.

### A. Constantes

Las declaramos con el siguiente formato:

```
<nombre_de_constante> CONSTANT <tipo> := <valor>;
```

Cuando declaramos una constante siempre se deberá asignar un valor. Por ejemplo:

```
Pct_iva CONSTANT REAL := 16;
```

Aunque PL/SQL no diferencia entre mayúsculas y minúsculas en los identificadores, sí lo hace en los literales y con el contenido de las variables y constantes (tal como ocurre en SQL).

### B. Literales

Los **literales** representan valores constantes directamente (sin recurrir a identificadores). Se utilizan para visualizar valores, hacer asignaciones a variables, constantes u otros objetos del programa. Pueden ser de varios tipos:

#### ● Carácter

Constan de un único carácter alfanumérico o especial introducido entre comillas simples. Ejemplos: 'A', 'a', 'j', '5', '%', '\*'.

Debemos ser muy cuidadosos con estos delimitadores, especialmente si utilizamos editores distintos del SQL\*Plus pues muchos editores inducen a confusión o tienden a cambiar estos signos.

#### ● Cadena

Son conjuntos de caracteres introducidos entre comillas simples. Ejemplos: 'Hola Mundo', 'Cliente N°: ', 'Introduzca un valor...'.

En ocasiones necesitaremos incluir el carácter utilizado como delimitador en la cadena o carácter que queremos representar. Por ejemplo: DBMS\_OUTPUT.PUT\_LINE('Peter''s hotel'); .

Dará el error: ORA-01756: quoted string not properly terminated.

En estos casos emplearemos dos comillas simples en el lugar donde debe aparecer una:

```
DBMS_OUTPUT.PUT_LINE('Peter''s hotel'); END;
```

El resultado será el deseado: Peter's hotel.

#### ● Numérico

Representan valores numéricos enteros o reales. Se pueden representar con notación científica. Ejemplos: 4, -7, 567.45, -458.456, 2.234E4, 5.25e-5, -8,75E+2



## 9. Fundamentos del lenguaje PL/SQL

### 9.6 Operadores y delimitadores

#### ◆ Booleano

Representan los valores lógicos verdadero, falso y nulo. Se representan mediante las palabras **TRUE**, **FALSE** y **NULL** escritas directamente en el código y sin comillas (no son literales sino palabras reservadas del lenguaje). Se pueden utilizar para asignar valores a variables y constantes, o para comprobar el resultado de expresiones lógicas. Ejemplos: `v_cobrado := TRUE;` `v_enviado := FALSE;`

#### ◆ Fecha/hora

Se utilizan para representar valores de fecha y hora según los distintos formatos estudiados anteriormente. Se representan mediante la expresión que indica el tipo DATE o TIMESTAMP seguida de la cadena delimitada que indica el valor que queremos representar. Por ejemplo: `DATE '2005-11-09'` `TIMESTAMP '2005-11-09 13:50:00'`.

También podemos representar valores correspondientes a los **subtipos TIMESTAMP WITH TIME ZONE** y **TIMESTAMP WITH LOCAL TIME ZONE** utilizando la palabra TIMESTAMP y la cadena en el formato correspondiente al subtipo que queremos representar como se muestra en los siguientes ejemplos:

`TIMESTAMP '2005-11-09 13:50:00 +06:00'` (para **TIMESTAMP WITH TIME ZONE**)

`TIMESTAMP '2005-11-09 13:50:00'` (para **TIMESTAMP WITH LOCAL TIME ZONE**)

## 9.6 Operadores y delimitadores

Los operadores se utilizan para asignar valores y formar expresiones. PL/SQL dispone de operadores de:

- **Asignación.** `:=` Asigna un valor a una variable.  
Por ejemplo: `Edad := 19;`
- **Concatenación.** `||` Une dos o más cadenas.  
Por ejemplo: `'buenos' || 'días'` dará como resultado `'buenosdías'`.
- **Comparación.** `=, !=, <, >, <=, >=, IN, IS NULL, LIKE, BETWEEN, ...`  
Funcionan igual que en SQL.
- **Aritméticos.** `+, -, *, /, **`. Se emplean para realizar cálculos. Algunos de ellos se pueden utilizar también con fechas.

`f1 - f2`. Devuelve el número de días que hay entre las fechas `f1` y `f2`.

`f + n`. Devuelve una fecha que es el resultado de sumar `n` días a la fecha `f`.

## 9. Fundamentos del lenguaje PL/SQL

### 9.6 Operadores y delimitadores



$f - n$ . Devuelve una fecha que es el resultado de restar  $n$  días a la fecha  $f$ .

- **Lógicos. AND, OR y NOT.** Permiten operar con expresiones que devuelven valores booleanos (comparaciones, etcétera). A continuación, se detallan las tablas de valores de los operadores lógicos AND, OR y NOT, teniendo en cuenta todos los posibles valores, incluida la ausencia de valor (NULL).

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

  

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

  

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

Tabla 9.2. Valores de los operadores lógicos.

- **Otros indicadores y delimitadores.**

( )	Delimitador de expresiones.
''	Delimitador de literales de cadena.
""	Delimitador de identificadores (utilización desaconsejable, en general).
<>>	Etiquetas.
/* */	Delimitador de comentarios de varias líneas.
--	Indicador de comentario de una línea.
%	Indicador de atributo (TYPE, ROWTYPE, FOUND,...).
:	Indicador de variables de transferencia ( <i>bind</i> ).
,	Separador de ítem de lista.
;	Terminador de instrucción.
@	Indicador de enlace de base de datos.

### A. Orden de precedencia en los operadores

El **orden de precedencia o prioridad de los operadores** determina el orden de evaluación de los operandos de una expresión. Por ejemplo, la siguiente expresión:  $12+18/6$ , dará como resultado 15, ya que la división se realizará primero. En la siguiente tabla se muestran los operadores disponibles agrupados y ordenados de mayor a menor, por orden de precedencia.

Prioridad	Operador	Operación
1	**, NOT	Exponenciación, negación.
2	*, /	Multiplicación, división.
3	+, -,	Suma, resta, concatenación.
4	=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparación.
5	AND	Conjunción.
6	OR	Disyunción.

Tabla 9.3. Orden de precedencia en los operadores.

Debemos tener cuidado con expresiones del tipo:

$100 > x > 0$  -- ilegal

Ya que primero se ejecutará  $100 > x$  y devolverá un valor que será VERDADERO, FALSO o NULL, y ése será el primer término de la siguiente comparación, lo cual causará un error. Para evitarlo, podemos indicar la expresión de la siguiente forma:

$(100 > x) AND (x > 0)$



## 9. Fundamentos del lenguaje PL/SQL

### 9.7 Funciones predefinidas

Aunque ésta es la prioridad establecida por defecto, podemos cambiarla utilizando paréntesis. Por ejemplo, en la expresión anterior, si queremos que la suma se haga antes que la división, lo indicaremos con la expresión:  $(12+18)/6$ . El resultado será 5.

Los operadores que se encuentran en el mismo grupo tienen la misma precedencia. En estos casos no se garantiza el orden de evaluación. Si queremos que se evalúen en algún orden concreto, deberemos utilizar paréntesis.

#### B. Evaluación en cortocircuito

PL/SQL da por concluida la evaluación de una expresión lógica cuando el resultado ha quedado determinado por la evaluación de una parte de ella. Esto ocurre cuando en una operación OR el primer operando es verdadero, o en una operación AND el primer operando es falso. De esta forma, los programas se ejecutan más rápidamente.

La evaluación en cortocircuito se puede aprovechar para evitar algunos problemas. Por ejemplo, suponemos que  $x$  e  $y$  son variables. Si queremos comprobar que  $y / x$  es mayor que 1, podemos evitar la posibilidad de un error ORA-01476 (intento de división por cero) aprovechando la evaluación en cortocircuito:

```
IF (x != 0) AND (y / x > 1) THEN ...
```

## 9.7 Funciones predefinidas

En PL/SQL se pueden utilizar todas las funciones de SQL. No obstante, algunas, como las de agrupamiento (AVG, MIN, MAX, COUNT, SUM, STDDEV, etcétera), solamente se pueden usar dentro de cláusulas de selección (SELECT).

Respecto a la utilización de funciones, también debemos tener en cuenta que:

- La función no modifica el valor de las variables o expresiones que se pasan como argumento, sino que devuelve un valor a partir de dicho argumento.
- Si a una función se le pasa un valor nulo en la llamada, normalmente devolverá un valor nulo.

Veamos un ejemplo de funciones: escribiremos un bloque PL/SQL que muestre la fecha y la hora con minutos y segundos.

```
SQL> BEGIN
 2      DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE,
 3 'DAY " " día " " DD " " de " MONTH " " de " YYYY " " a las " :HH24
 4 :MI:SS' ) );
 5  END;
 6 /
JUEVES   día  02  de MARZO    de 2006  a las :13:06:10
Procedimiento PL/SQL terminado con éxito.
```



## 9.8 Comentarios de documentación de los programas

Los comentarios se utilizan para documentar datos generales y particulares de los programas (el autor, la fecha, el objeto del programa, aspectos relevantes o hitos en el programa, función de determinados objetos o comentarios explicativos) e incluso para añadir legibilidad y organización a nuestros programas: líneas de separación, etcétera. En PL/SQL, podemos insertar comentarios en cualquier parte del programa. Estos comentarios pueden ser:

- **De línea con "--".** Todo lo que le sigue en esa línea será considerado comentario. Todos los comentarios incluidos en el código hasta el momento son de este tipo.
- **De varias líneas con "/\* <comentarios> \*/" (igual que en C).** Se pueden incluir en cualquier sección del programa. Es aconsejable, aunque no obligatorio, que incluyan una o varias líneas completas.

```
/* Este programa de prueba ha sido realizado por F. MONTERO
para documentar la realización de comentarios en PL/SQL
*/
```

```
BEGIN -- aquí comienza la sección ejecutable
```

## 9.9 Estructuras de control

PL/SQL dispone de estructuras para controlar el flujo de ejecución de los programas.

### ESTRUCTURAS ALTERNATIVAS:

#### Alternativa simple

```
IF <condicion> THEN
    instrucciones;
END IF;
```

Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN.

#### Alternativa doble

```
IF <condicion> THEN
    instrucciones1;
ELSE
    instrucciones2;
END IF;
```

Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN. En caso contrario, se ejecutarán las instrucciones que siguen a la cláusula ELSE.

#### Alternativa múltiple con ELSIF

```
IF <condicion1> THEN
    instrucciones1;
ELSIF <condicion2> THEN
    instrucciones2;
ELSIF <condicion3> THEN
    instrucciones3;
...
[ELSE
    instruccionesotras;] END IF;
```

Evaluá, comenzando desde el principio, cada condición, hasta encontrar alguna condición que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice, se ejecutará cuando no se ha cumplido ninguna de las condiciones anteriores.

La mayoría de las estructuras de control requieren evaluar una condición que en PL/SQL puede dar tres resultados: **TRUE**, **FALSE** o **NULL**. Pues bien, a efectos de estas estructuras, el valor **NULL** es equivalente a **FALSE**, es decir, se considerará que se cumple la condición sólo si el resultado es **TRUE**. En caso contrario (**FALSE** o **NULL**), se considerará que no se cumple.



## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control

La estructura CASE no está disponible en versiones anteriores a la 9i. En estos casos deberemos utilizar la alternativa múltiple ELSIF.

#### Alternativa múltiple con CASE de comprobación

```
CASE expresión
WHEN <valorcomprobac1> THEN
    instrucciones1;
WHEN <valorcomprobac2> THEN
    instrucciones2;
WHEN <valorcomprobac3> THEN
    instrucciones3;
...
[ELSE
    instruccionesotras;]
END CASE;
```

Calcula el resultado de la expresión que sigue a la cláusula CASE. A continuación, comprueba si el valor obtenido coincide con alguno de los valores especificados detrás de las cláusulas WHEN, en cuyo caso ejecutará la instrucción o instrucciones correspondientes. La cláusula ELSE es opcional; se ejecutará en caso de que no se encuentre un valor coincidente en las cláusulas WHEN precedentes.

#### Alternativa múltiple con CASE de búsqueda

```
CASE
WHEN <condicion1> THEN
    instrucciones1;
WHEN <condicion2> THEN
    instrucciones2;
WHEN <condicion3> THEN
    instrucciones3;
...
[ELSE
    instruccionesotras;]
END CASE;
```

Evaluá, comenzando desde el principio, cada condición, hasta encontrar alguna condición que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice, se ejecuta cuando no se ha cumplido ninguna de las condiciones anteriores.

Veamos un ejemplo:



#### Caso práctico

1 Supongamos que pretendemos modificar el salario de un empleado especificado en función del número de empleados que tiene a su cargo:

- Si no tiene ningún empleado a su cargo la subida será 50 €.
- Si tiene 1 empleado la subida será 80 €.
- Si tiene 2 empleados la subida será 100 €.
- Si tiene más de tres empleados la subida será 110 €.

Además, si el empleado es PRESIDENTE se incrementará el salario en 30 €.

```
DECLARE
    v_empleado_no NUMBER(4,0);          -- emple al que subir salario
    v_c_empleados NUMBER(2);            -- cantidad empl dependen de él
    v_aumento NUMBER(7) DEFAULT 0;      -- importe que vamos a aumentar.
    v_oficio VARCHAR2(10);
```

(Continúa)

## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control



(Continuación)

```
BEGIN
    v_empleado_no := &vt_empno;      -- var de sustitución lee n°emple

    SELECT oficio INTO v_oficio FROM emple
        WHERE emp_no = v_empleado_no;

    IF v_oficio = 'PRESIDENTE' THEN -- alternativa simple
        v_aumento := 30;
    END IF;

    SELECT COUNT(*) into v_c_empleados FROM emple
        WHERE dir = v_empleado_no;

    IF v_c_empleados = 0 THEN      -- alternativa múltiple
        v_aumento := v_aumento + 50;
    ELSIF v_c_empleados = 1 THEN
        v_aumento := v_aumento + 80;
    ELSIF v_c_empleados = 2 THEN
        v_aumento := v_aumento + 100;
    ELSE
        v_aumento := v_aumento + 110;
    END IF;

    UPDATE emple SET salario = salario + v_aumento WHERE emp_no = v_empleado_no;
    DBMS_OUTPUT.PUT_LINE(v_aumento);
END;
/
```

El resultado de la ejecución será:

```
Introduzca valor para vt_empno: 7839
140
```

En el programa anterior hemos utilizado una estructura ELSIF pero podíamos haber utilizado una estructura CASE en cualquiera de sus dos formatos:

#### CON CASE DE BÚSQUEDA

```
-----
CASE
WHEN v_c_empleados = 0 THEN
    v_aumento := v_aumento + 50;
WHEN v_c_empleados = 1 THEN
    v_aumento := v_aumento + 80;
WHEN v_c_empleados = 2 THEN
    v_aumento := v_aumento + 100;
ELSE
    v_aumento := v_aumento + 110;
END CASE;
```

#### CON CASE DE COMPROBACIÓN

```
-----
CASE v_c_empleados
WHEN 0 THEN
    v_aumento := v_aumento + 50;
WHEN 1 THEN
    v_aumento := v_aumento + 80;
WHEN 2 THEN
    v_aumento := v_aumento + 100;
ELSE
    v_aumento := v_aumento + 110;
END CASE;
```



## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control

En ocasiones, se puede usar NULL como una instrucción que no hace nada por motivos de claridad o facilidad de codificación:

```
CASE val
WHEN 1 THEN
    INSERT INTO TEMP VALUES 'UNO';
WHEN 2 THEN
    INSERT INTO TEMP VALUES 'DOS';
WHEN 0 THEN
    NULL;                                -- No hace nada.
ELSE THEN
    INSERT INTO TEMP VALUES 'ERROR'
END CASE;
```

### ESTRUCTURAS REPETITIVAS

#### Iterar

```
LOOP
    Instrucciones;
    EXIT [WHEN <condición>];
    instrucciones;
END LOOP;
```

Se trata de un bucle que se repetirá indefinidamente hasta que encuentre una instrucción EXIT sin condición o hasta que se cumpla la condición asociada a la cláusula EXIT WHEN. Es una **condición de salida**.

#### Mientras

```
WHILE <condicion> LOOP
    instrucciones;
END LOOP;
```

Se evalúa la condición y, si se cumple, se ejecutarán las instrucciones del bucle. El bucle se seguirá ejecutando mientras se cumpla la condición. Es una **condición de continuación**.

En un **bucle WHILE**, si la condición no se cumple al comienzo, no se ejecutará ni una sola línea pues la comprobación se hace antes de entrar en el bucle y posteriormente, una vez finalizadas todas las instrucciones que incluye. **LOOP**, sin embargo, siempre entrará en el bucle, ejecutará las primeras instrucciones y saldrá del bucle cuando se cumpla la condición.



#### Caso práctico

- 2 Supongamos que deseamos analizar una cadena que contiene los dos apellidos para guardar el primer apellido en una variable a la que llamaremos v\_1apel. Entendemos que el primer apellido termina cuando encontramos cualquier carácter distinto de los alfabéticos (en mayúsculas).

```
DECLARE
    v_apellidos VARCHAR2(25);
    v_1apel VARCHAR2(25);
    v_caracter CHAR;
    v_posicion INTEGER :=1;
```

(Continúa)

## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control



(Continuación)

```
BEGIN
    v_apellidos := '&vs_apellidos';

    v_caracter := SUBSTR(v_apellidos,v_posicion,1);
    WHILE v_caracter BETWEEN 'A' AND 'Z' LOOP
        v_lapel := v_lapel || v_caracter;
        v_posicion := v_posicion + 1;
        v_caracter := SUBSTR(v_apellidos,v_posicion,1);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('1er Apellido:'||v_lapel||'*');
END;
/
```

El resultado de la ejecución será:

```
Introduzca valor para vs_apellidos: GIL MORENO
1er Apellido:GIL*
Procedimiento PL/SQL terminado con éxito.
```

El mismo ejemplo con un bucle LOOP... END LOOP será:

```
DECLARE
    v_apellidos VARCHAR2(25);
    v_lapel VARCHAR2(25);
    v_caracter CHAR;
    v_posicion INTEGER :=1;
BEGIN
    v_apellidos := '&vs_apellidos';

    -- desaparece la asignación de v_caracter antes del bucle
    -- se asignará dentro al comienzo del bucle.
    LOOP
        v_caracter := SUBSTR(v_apellidos,v_posicion,1);
        EXIT WHEN v_caracter NOT BETWEEN 'A' AND 'Z';
        v_lapel := v_lapel || v_caracter;
        v_posicion := v_posicion + 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('1er Apellido:'||v_lapel||'*');
END;
/
```

**EXIT** es una instrucción en sí misma (por eso lleva punto y coma al final) y puede ser utilizada con o sin la cláusula WHEN.

```
LOOP
    instrucciones;
    IF <condición> THEN
```



## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control

```
        EXIT;
    END IF;
    instrucciones;
END LOOP;
```

PL/SQL permite la utilización de EXIT incluso en otros bucles y estructuras. Esta práctica es totalmente desaconsejable.

#### Para

```
FOR <variablecontrol> IN <valorInicio>..<valorFinal> LOOP
    instrucciones;
END LOOP;
```

Donde <variablecontrol> es la variable de control del bucle que se declara de manera implícita como variable local al bucle de tipo BINARY\_INTEGER. Esta variable tomará, en primer lugar, el valor especificado en la expresión numérica <valorInicio>, incrementándose en uno para cada nueva iteración hasta alcanzar el valor especificado en la expresión numérica <valorFinal>.

Se utiliza esta estructura cuando se conoce o se puede conocer a priori el número de veces que se debe ejecutar un bucle.

El incremento siempre es una unidad, pero puede ser negativo utilizando la **opción REVERSE**:

```
FOR <variable> IN REVERSE <valorFinal>..<valorInicio>
LOOP
    instrucciones;
    ...
END LOOP;
```

En este caso, comenzará por el valor especificado en segundo lugar e irá restando una unidad en cada iteración:

```
SQL> BEGIN
  2      FOR i IN REVERSE 1..3 LOOP
  3          DBMS_OUTPUT.PUT_LINE(i);
  4      END LOOP;
  5  END;
  6  /
3
2
1
Procedimiento PL/SQL terminado con éxito.
```

Cuando empleamos la opción REVERSE, comenzará por el segundo valor hasta tomar el que sea igual o menor que el especificado en primer lugar. El bloque que se muestra a continuación tiene un error de diseño, ya que no llega siquiera a entrar en el bucle:

## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control



```
SQL> BEGIN
 2   FOR i IN REVERSE 5..1 LOOP    -- ERROR
 3     DBMS_OUTPUT.PUT_LINE(i);
 4   END LOOP;
 5 END;
 6 /
```

Procedimiento PL/SQL terminado con éxito.

Podemos indicar los valores mínimo y máximo mediante expresiones:

```
DECLARE
  Num1 INTEGER;
  Num2 INTEGER;
  ...
BEGIN
  ...
  FOR x IN Num1..Num2 LOOP
    ...
  END LOOP;
  ...
END;
```

PL/SQL no dispone de la **opción STEP** que tienen otros lenguajes, la cual permite especificar incrementos distintos de uno.

Respecto a la variable de control, hay que tener en cuenta que:

- No hay que declararla.
- Es local al bucle y no es accesible desde el exterior del bucle, ni siquiera en el mismo bloque.
- Se puede usar dentro del bucle en una expresión, pero no se le pueden asignar valores.

En el siguiente ejemplo, definimos una variable e intentamos usarla como variable de control. Aun en ese caso la estructura creará la suya propia como local, quedando la nuestra como global en el bucle:

```
<<ppal>>
DECLARE
  i INTEGER;
BEGIN
  ...
  FOR i IN 1..10 LOOP
    ...
  /* cualquier referencia a i será entendida como a la
  variable local al bucle. Si quisieramos referirnos a
  la otra lo debemos hacer como ppal.i */
  END LOOP;
  ...
  /*La variable local del bucle ya no existe aquí */
END ppal;
```

Veamos algunos ejemplos de aplicación:



## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control



### Caso práctico

- 3 Vamos a construir de dos maneras un bloque PL/SQL que escriba la cadena 'HOLA' al revés.

#### Utilizando un bucle FOR

```
SQL> DECLARE
 2   r_cadena VARCHAR2(10);
 3 BEGIN
 4   FOR i IN REVERSE 1..LENGTH('HOLA') LOOP
 5     r_cadena := r_cadena||SUBSTR('HOLA',i,1);
 6   END LOOP;
 7   DBMS_OUTPUT.PUT_LINE(r_cadena);
 8* END;
SQL> /
ALOH
```

Procedimiento PL/SQL terminado con éxito.

#### Utilizando un bucle WHILE

```
SQL> DECLARE
 2   r_cadena VARCHAR2(10);
 3   i BINARY_INTEGER;
 4 BEGIN
 5   i := LENGTH('HOLA');
 6   WHILE i >= 1 LOOP
 7     r_cadena=r_cadena||SUBSTR('HOLA',i,1);
 8     i := i - 1;
 9   END LOOP;
10   DBMS_OUTPUT.PUT_LINE(r_cadena);
11* END;
SQL> /
ALOH
```

Procedimiento PL/SQL terminado con éxito.



### Actividades propuestas

- 2 Escribe un bloque PL/SQL que realice la misma función del ejemplo anterior pero usando un bucle ITERAR.

### A. Utilización de etiquetas

Las **etiquetas** sirven para marcar o nombrar determinadas partes del código del programa como bloques, estructuras de control y otras.

Podemos situar etiquetas en nuestros programas utilizando el formato:

```
<<nombreetiqueta>>
```

Donde los delimitadores `<< >>` forman parte de la sintaxis y `nombretiqueta` representa un identificador válido PL/SQL que utilizaremos después (en este caso, sin los delimitadores) para hacer referencia al elemento.

Podemos etiquetar bucles y otras estructuras para conseguir mayor legibilidad:

```
<<mibucle>>
LOOP
  instrucciones;
  ...
END LOOP mibucle;
```

## 9. Fundamentos del lenguaje PL/SQL

### 9.9 Estructuras de control



También se pueden etiquetar las estructuras para eliminar ambigüedades, hacer visibles variables globales y conseguir otras funcionalidades:

```
<<bucleexterno>>
LOOP
...
LOOP
...
EXIT bucleexterno WHEN ... -- sale de ambos bucles
END LOOP;
END LOOP bucleexterno;
```

En PL/SQL se puede usar la instrucción **GOTO etiqueta**. Para poder utilizar esta orden se deben cumplir las siguientes condiciones:

- No puede haber otra etiqueta en el entorno actual con el mismo nombre.
- La etiqueta debe preceder a un bloque o a un conjunto de órdenes ejecutables.
- La etiqueta no puede estar dentro de un IF, de un LOOP ni de un SUB-BLOQUE internos al bloque donde se produce la llamada.
- Desde una excepción no se puede pasar el control del programa a una etiqueta que está en otra sección del mismo bloque.

#### Ejemplos de usos correctos

```
BEGIN
...
GOTO insertar_fila;
...
<<insertar_fila>>
INSERT INTO empleados VALUES ...
END;

También es posible:

BEGIN
...
<<insertar_fila>>
BEGIN
    INSERT INTO empleados VALUES ...
    ...
END;
...
GOTO insertar_fila;
...
END;
```

#### Ejemplos de usos ilegales

```
BEGIN
...
FOR i IN 1..10 LOOP
...
GOTO fin_loop;
...
<<fin_loop>> --illegal, no hay instrucciones ejecutables
END LOOP;
...
GOTO insertar_fila;
IF ... THEN
...
<<insertar_fila>>
    INSERT INTO empleados VALUES ... --illegal, está en un if
...
END IF;
...
GOTO otro_sub; -- illegal, está en otro sub-bloque
...
BEGIN
...
<<otro_sub>>
    DELETE FROM ...
END;
...
<<mi_etiqueta>>
...
EXCEPTION
    WHEN ... THEN
        GOTO mi_etiqueta; --illegal está en el bloque actual.
END;
...
```



## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones

## 9.10 Subprogramas: procedimientos y funciones

Los **subprogramas** son bloques PL/SQL que tienen un nombre y pueden recibir y devolver valores. Normalmente se guardan en la base de datos y podemos ejecutarlos invocándolos desde otros subprogramas o herramientas.

En todo subprograma podemos distinguir:

- **La cabecera o especificación del subprograma**, que contiene:
  - Nombre del subprograma.
  - Los parámetros con sus tipos (opcional).
  - Tipo de valor de retorno (en el caso de las funciones).
- **El cuerpo del subprograma**. Es un bloque PL/SQL que incluye:
  - Declaraciones (opcional).
  - Instrucciones.
  - Manejo de excepciones (opcional).

En PL/SQL podemos distinguir dos tipos de subprogramas: *procedimientos* y *funciones*. Veámoslos:

### A. Procedimientos

Tienen la siguiente estructura general:

```
PROCEDURE <nombreprocedimiento>
[ ( <lista de parámetros> ) ]
IS
    [<declaraciones>]
BEGIN
    <instrucciones>;
[EXCEPTION
    <excepciones>]
END [<nombreprocedimiento>];
```

Podemos apreciar dos partes:

- **La cabecera o especificación del procedimiento.** Comienza con la palabra PROCEDURE y termina después de la declaración de parámetros.
- **El cuerpo del procedimiento.** Corresponde con un bloque PL/SQL. Comienza a continuación de la palabra IS (o AS) y termina con la palabra END, opcionalmente seguida del nombre del procedimiento. En el formato genérico anterior corresponde a la zona sombreada.

## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones



Para crear un procedimiento desde SQL\*Plus usaremos el siguiente formato:

```
CREATE [OR REPLACE] PROCEDURE <nombreprocedimiento>
[(listadeparametros)]
AS ...
```

A continuación, se introducirá el bloque de código PL/SQL sin la palabra DECLARE.

La **opción REPLACE** actúa en el caso de que hubiese un subprograma almacenado con ese nombre, sustituyéndolo por el nuevo.

En la lista de parámetros se encuentra la declaración de cada uno de los parámetros que se utilizan para pasar valores al programa separados por comas:

```
(nombreparam1 TIPOP1, nombreparam2 TIPOP2, nombreparam3
TIPOP3, ...)
```

Por concordancia gramatical se utilizará AS en lugar de IS al crear un procedimiento o función con la orden CREATE OR REPLACE. Pero, a efectos del compilador, se puede utilizar cualquiera de las dos.

#### Caso práctico

- 4 Crearemos un procedimiento que reciba un número de empleado y una cadena correspondiente a su nuevo oficio. El procedimiento deberá localizar el empleado, modificar el oficio y visualizar los cambios realizados.

```
CREATE OR REPLACE PROCEDURE cambiar_oficio (
    num_empleado NUMBER,          -- En los parámetros ..
    nuevo_oficio VARCHAR2)        -- ..no se especifica tamaño
AS
    v_anterior_oficio emple.oficio%TYPE;
BEGIN
    SELECT oficio INTO v_anterior_oficio FROM emple
    WHERE emp_no = num_empleado;

    UPDATE emple SET oficio = nuevo_oficio
    WHERE emp_no = num_empleado;
    DBMS_OUTPUT.PUT_LINE(num_empleado||'*Oficio Anterior:'||v_anterior_oficio||
                           '*Oficio Nuevo :'||nuevo_oficio );
END cambiar_oficio;
/
```

El sistema responderá:

Procedimiento creado.

Ahora el procedimiento está creado y almacenado en la base de datos. Para ejecutarlo podemos invocar el procedimiento desde cualquier herramienta de Oracle, por ejemplo, desde SQL\*Plus:

```
SQL> EXECUTE CAMBIAR_OFICIO(7902,'DIRECTOR');
7902*Oficio Anterior:ANALISTA*Oficio Nuevo :DIRECTOR
Procedimiento PL/SQL terminado con éxito.
```



## 9. Fundamentos del lenguaje PL/SQL

9.10 Subprogramas: procedimientos y funciones



### Actividades propuestas

- 3 Escribe un procedimiento con funcionalidad similar al ejemplo anterior, que recibirá un número de empleado y un número de departamento y asignará al empleado el departamento indicado en el segundo parámetro.

Podemos invocar al procedimiento desde otro bloque, procedimiento o función. Para ello escribiremos el nombre del procedimiento seguido de la lista de parámetros entre paréntesis, y de un punto y coma:

```
nombreprocedimientoalquellamamos (listadeparametros);
```

La llamada a un procedimiento es una instrucción por sí misma. Cuando se produce la llamada, el control pasa al procedimiento llamado hasta que finaliza su ejecución y el control retorna a la línea siguiente a la llamada.

Por ejemplo, podemos crear un bloque que reciba el apellido y el oficio nuevo. El programa buscará el número de empleado y utilizará una llamada al procedimiento anterior para cambiar oficio:

```
CREATE OR REPLACE PROCEDURE cam_ofi_ (
    v_apellido VARCHAR,
    nue_oficio VARCHAR2)
IS
    v_n_empleado emple.emp_no%TYPE;
BEGIN
    SELECT emp_no INTO v_n_empleado FROM emple
        WHERE apellido = v_apellido;
    cambiar_oficio(v_n_empleado, nue_oficio);
END cam_ofi_;
```

No es obligatorio escribir el nombre de subprograma detrás del END, pero es aconsejable por razones de legibilidad.

La ejecución del nuevo programa será:

```
SQL> EXECUTE cam_ofi_('FERNANDEZ','ANALISTA');
7902*Oficio Anterior:DIRECTOR*Oficio Nuevo :ANALISTA
Procedimiento PL/SQL terminado con éxito.
```

## B. Funciones

Las **funciones** tienen una estructura y funcionalidad similar a los procedimientos pero, a diferencia de éstos, las funciones devuelven siempre un valor:

```
FUNCTION <nombrefunción>
    [<lista de parámetros>]
RETURN <tipo de valor devuelto >
IS
```

## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones



```
[<declaraciones>]  
BEGIN  
    <instrucciones>;  
    RETURN <expresión>;  
    ...  
[EXCEPTION  
    <excepciones>]  
END [<nombrededefunción>];
```

La lista de parámetros es opcional. Si no hay parámetros no debemos poner paréntesis pues dará error igual que en los procedimientos. Sin embargo, es obligatorio el uso de la cláusula **RETURN** en la cabecera y el comando **RETURN** en el cuerpo del programa. No debemos confundirlas:

- La cláusula **RETURN** de la cabecera especifica el tipo del valor que retorna la función.
- En el cuerpo del programa, el comando **RETURN** devuelve el control al programa que llamó a la función, asignando el valor de la expresión que sigue al **RETURN** a la variable que figura en la llamada a la función.

Antes de ejecutar un subprograma almacenado, Oracle marca un punto de salvaguarda implícito, de forma que si el subprograma falla durante la ejecución, se desharán todos los cambios realizados por él.

De manera análoga a los procedimientos, para crear o modificar una función utilizaremos el comando **CREATE OR REPLACE FUNCTION**:

```
CREATE OR REPLACE FUNCTION Encontrar_Num_empleado (  
    v_apellido VARCHAR2)  
RETURN REAL  
AS  
    N_empleado emple.emp_no%TYPE;  
BEGIN  
    SELECT emp_no INTO N_empleado FROM emple  
        WHERE apellido = v_apellido;  
    RETURN N_empleado;  
END Encontrar_num_empleado;
```

Un procedimiento también puede usar la cláusula **RETURN** (en este caso, sin devolver ningún valor) para devolver el control al programa que lo llamó, pero no es una técnica recomendable.

El formato de llamada a una función consiste en utilizarla como parte de una expresión:

```
<variable> := <nombrededefunción>[(listadeparámetros)];
```

Para invocar a una función desde SQL también tenemos que «hacer algo» con el valor que devuelve, por ejemplo, utilizarlo como parámetro para otra llamada:

```
SQL> BEGIN DBMS_OUTPUT.PUT_LINE(ENCONTRAR_NUM_EMPLEADO  
                                ('GIL')) ;END;  
2 /  
7788  
Procedimiento PL/SQL terminado con éxito.
```

Una función puede tener varios **RETURN** pero solo se ejecutará uno de ellos en cada llamada a la función:



## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones

```
...
IF nota < 5 THEN
    RETURN 'SUSPENSO';
ELSE
    RETURN 'APROBADO';
END IF;
...
```

## C. Parámetros

Los subprogramas utilizan parámetros para pasar y recibir información. Hay dos clases:

- **Parámetros actuales o reales.** Son las variables o expresiones indicadas en la llamada a un subprograma.
- **Parámetros formales.** Son variables declaradas en la especificación del subprograma.

Las declaraciones de variables locales se hacen después del IS, que equivale al DECLARE. En este caso, sí se deberá indicar la longitud pues ya no se trata de parámetros.

Si es necesario, PL/SQL hará la conversión automática de tipos; sin embargo, los tipos de los parámetros actuales y los correspondientes parámetros formales deben ser compatibles.

Podemos hacer el paso de parámetros utilizando la *notación posicional*, *nominal* o *mixta* (ambas):

- **Notación posicional:** El compilador asocia los parámetros actuales a los formales basándose en su posición.
- **Notación nominal:** El símbolo => después del parámetro actual y antes del nombre del formal indica al compilador la correspondencia.
- **Notación mixta:** Consiste en usar ambas notaciones con la restricción de que la notación posicional debe preceder a la nominal.

Por ejemplo, dada la siguiente especificación del procedimiento ges\_dept:

```
PROCEDURE ges_dept (
    N_departamento INTEGER,
    Localidad VARCHAR2
IS...
```

Desde el siguiente bloque se podrán realizar las llamadas indicadas:

```
DECLARE
    Num_dep INTEGER;
    Local   VARCHAR(14)
BEGIN
    ...
    -- posicional  ges_dept(Num_dep, local);
    -- nominal      ges_dept(Num_dep => N_departamento,
                           local => localidad);
```

## 9. Fundamentos del lenguaje PL/SQL

9.10 Subprogramas: procedimientos y funciones



```
-- nominal      ges_dept(local => localidad, Num_dep =>
-- mixta        N_departamento);
               ges_dept(Num_dept, Local => localidad);
...
END;
```

### Actividades propuestas

#### 4 Dado el siguiente procedimiento:

```
PROCEDURE crear_depart (
    v_num_dept depart.dept_no%TYPE,
    v_dnombre depart.dnombre%TYPE DEFAULT 'PROVISIONAL',
    v_loc      depart.loc%TYPE DEFALUT 'PROVISIONAL')
IS
BEGIN
    INSERT INTO depart
        VALUES (v_num_dept, v_dnombre, v_loc);
END crear_depart;
```

Indica cuáles de las siguientes llamadas son correctas y cuáles incorrectas. En el caso de que sean incorrectas, escribe la llamada correcta usando la notación posicional, siempre que sea posible:

```
crear_depart;
crear_depart(50);
crear_depart('COMPRAS');
crear_depart(50, 'COMPRAS');
crear_depart('COMPRAS', 50);
crear_depart('COMPRAS', 'VALENCIA');
crear_depart(50, 'COMPRAS', 'VALENCIA');
crear_depart('COMPRAS', 50, 'VALENCIA');
crear_depart('VALENCIA', 'COMPRAS');
crear_depart('VALENCIA', 50);
```

PL/SQL soporta tres tipos de parámetros:

Tipo	Características y utilización
IN	Son parámetros de <b>ENTRADA</b> ; se usan para pasar valores al subprograma. Dentro del subprograma el parámetro actúa como una constante, es decir, no se le puede asignar ningún valor. Por tanto, se sitúa siempre a la derecha del operador de asignación. El parámetro actual puede ser una variable, constante, literal o expresión.
OUT	Son parámetros de <b>SALIDA</b> ; se usan para devolver valores al programa que hizo la llamada. Dentro del subprograma, el parámetro actúa como una variable no inicializada y no puede intervenir en ninguna expresión, salvo para tomar un valor. Se sitúa siempre a la izquierda del operador de asignación.
IN OUT	Son parámetros de <b>ENTRADA/SALIDA</b> ; permiten pasar un <b>valor inicial</b> y devolver un <b>valor actualizado</b> . Dentro del subprograma actúa como una variable inicializada. Puede intervenir en otras expresiones y puede tomar nuevos valores. El <b>parámetro actual</b> debe ser una variable.



## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones

No podemos asignar valores nuevos a los parámetros formales de entrada pues nos encontraremos con el error:

PLS-00363: expression 'VParametro' cannot be used as an assignment target.

Si queremos cambiar el valor de un parámetro formal de entrada deberemos declararlo de tipo IN OUT.

El formato genérico de la declaración de cada uno de los parámetros es:

```
<nombrevariable> [ IN | OUT | IN OUT ] < tipodedato>
[ { := | DEFAULT } <valor> ]
```

Debemos tener en cuenta, además, las siguientes reglas:

- Al indicar los parámetros debemos especificar el tipo, pero no el tamaño.
- En el caso de que el subprograma no tenga parámetros no se pondrán los paréntesis.
- Cuando un subprograma recibe un parámetro en modo OUT y se produce una excepción no tratada, el parámetro actual correspondiente queda sin ningún valor.

**Valores por defecto en el paso de parámetros de entrada (modo IN):** los parámetros de entrada (todos los que hemos manejado hasta este momento) se pueden inicializar con valores por omisión, es decir, indicando al subprograma que en el caso de que no se pase el parámetro correspondiente, asuma un valor por defecto. Para ello, se utiliza la opción DEFAULT <valor>, o bien := <valor>.



### Caso práctico

5 Supongamos que nos han solicitado un programa de cambio de divisas para un banco que cumpla las siguientes especificaciones:

- Recibirá una cantidad en euros y el cambio (divisas/euro) de la divisa.
- También podrá recibir una cantidad correspondiente a la comisión que se cobrará por la transacción. En el caso de que no reciba dicha cantidad el programa calculará la comisión que será de un 0,2% del importe, con un mínimo de 3 euros.
- El programa calculará la comisión, la deducirá de la cantidad inicial y calculará el cambio en la moneda deseada, retornando estos dos valores (comisión y cambio) a los parámetros actuales del programa que realice la llamada para solicitar el cambio de divisas.

```
CREATE OR REPLACE PROCEDURE cambiar_divisas (
    cantidad_euros      IN      NUMBER, -- parámetro entrada
    cambio_actual        IN      NUMBER, -- parámetro entrada
    cantidad_comision   IN OUT  NUMBER, -- parámetro de e/s
    cantidad_divisas    OUT     NUMBER) -- parámetro de salida
AS
    pct_comision      CONSTANT NUMBER (3,2) := 0.2;
    minimo_comision  CONSTANT NUMBER (6) DEFAULT 3;
BEGIN
    IF cantidad_comision IS NULL THEN
        cantidad_comision := GREATEST(cantidad_euros/100*pct_comision,
                                         minimo_comision);
    END IF;
    cantidad_divisas := cantidad_euros / cambio_actual;
    cantidad_comision := cantidad_comision;
END;
```

(Continúa)

## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones



(Continuación)

```
END IF;
cantidad_divisas := (cantidad_euros - cantidad_comision) * cambio_actual;
END;
/
```

Una vez creado el procedimiento podremos diseñar programas que hagan uso de él teniendo en cuenta que los parámetros formales para llamar al programa deberán ser cuatro. De éstos, los dos últimos deberán ser variables, que recibirán los valores de la ejecución del programa, tal como aparece en el siguiente procedimiento:

```
CREATE OR REPLACE PROCEDURE mostrar_cambio_divisas (
    eur NUMBER,
    cambio NUMBER)
AS
    v_comision NUMBER (9);
    v_divisas NUMBER (9);
BEGIN
    Cambiar_divisas(eur, cambio, v_comision, v_divisas);
    DBMS_OUTPUT.PUT_LINE ('Euros           : ' ||
                           TO_CHAR( eur, '999,999,999.999'));
    DBMS_OUTPUT.PUT_LINE ('Divisas X 1 euro : ' ||
                           TO_CHAR( cambio, '999,999,999.999'));
    DBMS_OUTPUT.PUT_LINE ('Euros Comisión   : ' ||
                           TO_CHAR( v_comision, '999,999,999.999'));
    DBMS_OUTPUT.PUT_LINE ('Cantidad divisas : ' ||
                           TO_CHAR( v_divisas, '999,999,999.999'));
END;
/
```

Llamamos al programa pasándole la cantidad y el cambio respecto al euro de la divisa queremos cambiar a euros.

```
SQL> EXECUTE MOSTRAR_CAMBIO_DIVISAS(2500, 1.220);
Euros           :      2,500.000
Divisas X 1 euro :        1.220
Euros Comisión   :         5.000
Cantidad divisas :     3,044.000
```

Procedimiento PL/SQL terminado con éxito.

## D. Subprogramas almacenados

Los subprogramas (procedimientos y funciones) que hemos visto hasta ahora se pueden compilar independientemente y almacenar en la base de datos Oracle.

Cuando creamos procedimientos o funciones almacenados desde SQL\*Plus utilizando los comandos CREATE PROCEDURE o CREATE FUNCTION, Oracle automáticamente compila el código fuente, genera el código objeto (llamado *P-código*) y los guarda en el diccionario de datos. De este modo, quedan disponibles para su utilización.

Oracle dispone de opciones avanzadas y funcionalidades especiales para el paso de parámetros que pueden resultar interesantes para el manejo de estructuras complejas. Estas opciones exceden del objetivo de este libro.



## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones

Los programas almacenados tienen dos estados: *disponible (valid)* y *no disponible (invalid)*. Si alguno de los objetos referenciados por el programa ha sido borrado o alterado desde la última compilación del programa, quedará en situación de «no disponible» y se compilará de nuevo automáticamente en la próxima llamada. Al compilar de nuevo, Oracle determina si hay que compilar algún otro subprograma referido por el actual. Se puede producir una cascada de compilaciones.

Estos estados se pueden comprobar en la vista **USER\_OBJECTS**:

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS FROM
      USER_OBJECTS WHERE OBJECT_NAME='CAMBIAR_OFICIO';
```

OBJECT_NAME	OBJECT_TYPE	STATUS
CAMBIAR_OFICIO	PROCEDURE	VALID

Podemos acceder al código fuente almacenado mediante la vista **USER\_SOURCE**:

```
SQL> SELECT LINE, SUBSTR(TEXT,1,60) FROM USER_SOURCE WHERE
      NAME = 'CAMBIAR_OFICIO';
      LINE SUBSTR(TEXT,1,60)
-----
1  PROCEDURE cambiar_oficio (
2      num_empleado NUMBER,      -- En los parámetros ..
3      nuevo_oficio VARCHAR2)  -- ..no se especifica tamaño
4  AS
5      v_anterior_oficio emple.oficio%TYPE;
6  BEGIN
7      SELECT oficio INTO v_anterior_oficio FROM emple
8      WHERE emp_no = num_empleado;
9
10     UPDATE emple SET oficio = nuevo_oficio
11         WHERE emp_no = num_empleado;
12     DBMS_OUTPUT.PUT_LINE(num_empleado||
13         '*Oficio Anterior:'||v_anterior_oficio||
14         '*Oficio Nuevo   :'||nuevo_oficio );
15 END cambiar_oficio;
```

Para volver a compilar un subprograma almacenado en la base de datos se emplea la orden **ALTER** con la opción **COMPILE**, indicando **PROCEDURE** o **FUNCTION**, según el tipo de subprograma:

```
ALTER {PROCEDURE|FUNCTION} nombre_subprograma COMPILE;
```

Para borrar un subprograma, igual que para eliminar otros objetos, se usa la orden **DROP** seguida del tipo de subprograma (**PROCEDURE** o **FUNCTION**):

```
DROP {PROCEDURE|FUNCTION} nombre_subprograma;
```

## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones



#### E. Subprogramas locales

Los **subprogramas locales** son procedimientos y funciones que se crean dentro de otro subprograma o de un bloque, al final de la sección declarativa:

```
CREATE OR REPLACE pr_ejem1 /* programa ppal. (contiene el
local) */
AS
... /* lista de declaraciones: variables, etc. */

PROGRAM sprloc1 /* comienza el subprograma local */
...
IS
...
/*declaraciones locales al subprograma local */
BEGIN
...
/*instrucciones del subprograma local */
END;

BEGIN
...
sprloc1; /* llamada al subprograma local */
...
END pr_ejem1;
```

Estos subprogramas tienen las siguientes particularidades:

- Se declaran al final de la sección declarativa de otro subprograma o bloque.
- Sólo están disponibles en el bloque en que se crearon y los bloques hijos de éste. Se les aplica las mismas reglas de ámbito y visibilidad que a las variables declaradas en el mismo bloque.
- Se utilizará este tipo de subprogramas cuando no se contemple su reutilización por otros subprogramas (distintos de aquel en el que se declaran).
- En el caso de subprogramas locales con referencias cruzadas o de subprogramas mutuamente recursivos, hay que realizar *declaraciones anticipadas*, tal como se explica a continuación.

PL/SQL necesita que todos los identificadores, incluidos los subprogramas, estén declarados antes de usarlos. Por eso, la llamada que aparece en el siguiente subprograma es ilegal:

```
DECLARE
...
PROCEDURE subprograma1 ( ... ) IS
BEGIN
...
Subprograma2( ... ); -- > ERR. identificador no
declarado
...
END;
```

Las **declaraciones anticipadas** permiten definir subprogramas en el orden que queramos, incluso programas mutuamente recursivos.



## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones

```
PROCEDURE subprograma2 ( ... ) IS
BEGIN
    ...
END;
...
```

Se podía haber invertido el orden de declaración de los procedimientos anteriores, lo cual hubiera resuelto este caso, pero no sirve para procedimientos mutuamente recursivos. El problema se resuelve utilizando declaraciones anticipadas de subprogramas.

```
DECLARE
    PROCEDURE subprograma2 (...); -- declaración anticipada
        PROCEDURE subprograma1 ( ... ) IS
    BEGIN
        Subprograma2( ... );      -- > ahora no dará error.
        ...
    END;
    PROCEDURE subprograma2 ( ... ) IS
    BEGIN
        ...
    END;
    ...
```

## F. Recursividad

No se deben usar algoritmos recursivos en conjunción con cursosres FOR LOOP, ya que se puede exceder el número máximo de cursosres abiertos. Siempre se pueden sustituir las estructuras recursivas por bucles.

PL/SQL implementa, al igual que la mayoría de los lenguajes de programación, la posibilidad de escribir subprogramas recursivos:

```
CREATE OR REPLACE FUNCTION factorial
    (n POSITIVE)
    RETURN INTEGER                      -- > devuelve n!
    AS
    BEGIN
        IF n = 1 THEN                  -- > condición de terminación
            RETURN 1;
        ELSE
            RETURN n * factorial(n - 1); -- > llamada recursiva
        END IF;
    END factorial;
```

## G. Sobrecarga en los nombres de subprogramas

Teniendo en cuenta las conversiones automáticas y la posibilidad de valores por defecto, la sobrecarga de nombres de programas se debe usar solamente cuando hay total seguridad.

PL/SQL permite sobrecarga en los nombres de subprogramas dentro de paquetes (los veremos más adelante) siempre que los parámetros formales de los subprogramas difieran en número, orden y/o tipo de dato (familia de tipo de dato). Por ejemplo, podemos crear las variables:

- **buscar\_emple** (NUMBER) que recibe un número de empleado y lo busca en la base de datos.

## 9. Fundamentos del lenguaje PL/SQL

### 9.10 Subprogramas: procedimientos y funciones



- `buscar_emple(VARCHAR2)` recibe un nombre, lo busca y muestra los datos encontrados.

En realidad, son dos procedimientos distintos (aunque tengan similitudes, el código es distinto). Oracle los diferencia en las llamadas (y otros comandos) por el contexto, en este caso por los parámetros.



## 9. Fundamentos del lenguaje PL/SQL

### Conceptos básicos



## Conceptos básicos

- Los **tipos de datos escalares** estudiados son: *carácter* (CHAR, NCHAR, VARCHAR2, NVARCHAR2...), *numérico* (NUMBER, BINARY\_INTEGER, PLS\_INTEGER), *booleano*, *Fecha/hora* (DATE, TIMESTAMP...) y otros ( ROWID, UROWID,...).
- Los **identificadores** en PL/SQL pueden tener entre 1 y 30 caracteres de longitud; el primer carácter debe ser una letra y los restantes deben ser caracteres alfanuméricos o signos admitidos (letras, dígitos, los signos de dólar, almohadilla y subguion); no pueden incluir signos de puntuación, espacios, etcétera.
- El **formato para declarar una variable** es:  
`<nombre_de_variable> <tipo> [NOT NULL]  
[ { := | DEFAULT} <valor>]`
- Las variables se crean al comienzo del bloque y dejan de existir una vez finalizada la ejecución del bloque en el que han sido declaradas. El ámbito de una variable incluye el bloque en el que se declara y los bloques «hijos» de este.
- Al declarar una constante deberemos incluir CONSTANT y asignar un valor.
- En PL/SQL, podemos insertar comentarios: de línea con -- o de varias líneas con /\* ... \*/.
- Las **estructuras de control** son:
  - Alternativa simple:* IF ...THEN ...; END IF;
  - Alternativa doble:* IF...THEN ...; ELSE...; END IF;
  - Alternativa múltiple:* IF... THEN ...; ELSIF ... THEN ...; ELSIF ... THEN ...; ... ELSE ...; END IF;
  - Alternativa multiple:* CASE ... WHEN ... THEN ...; WHEN ... THEN ...; ... ELSE ...; END CASE;
  - Iterar:* LOOP ... ; EXIT WHEN ... ; END LOOP;
  - Mientras:* WHILE ... LOOP ...; END LOOP;  
Para: FOR I in 1.. n LOOP ...; END LOOP;

- Para crear un *procedimiento* o *función* desde SQL\*Plus usaremos CREATE OR REPLACE ...

```
PROCEDURE <nombaprocedimiento>
    [ ( <lista de parámetros> ) ]
IS   ... < BLOQUE PL/SQL > ;

FUNCTION <nombrefunción>
    [ ( <lista de parámetros> ) ]
RETURN <tipo d valor devuelto >
IS   ... < BLOQUE PL/SQL que incluye
      RETURN <expresión>;
```

- La instrucción para invocar un procedimiento es un comando en sí misma. En el caso de las funciones es una expresión que debe hacer algo con el valor devuelto.
- Al indicar los parámetros debemos especificar el tipo pero no el tamaño. En el caso de que el subprograma no tenga parámetros no se pondrán los paréntesis.
- Los parámetros en modo IN, dentro del subprograma, actúan como una constante, es decir, no se les puede asignar ningún otro valor.



# Actividades complementarias



- 1 Escribe un procedimiento que reciba dos números y visualice su suma.
- 2 Codifica un procedimiento que reciba una cadena y la visualice al revés.
- 3 Reescribe el código de los dos ejercicios anteriores para convertirlos en funciones que retornen los valores que mostraban los procedimientos.
- 4 Escribe una función que reciba una fecha y devuelva el año, en número, correspondiente a esa fecha.
- 5 Escribe un bloque PL/SQL que haga uso de la función anterior.
- 6 Desarrolla una función que devuelva el número de años completos que hay entre dos fechas que se pasan como parámetros.
- 7 Escribe una función que, haciendo uso de la función anterior, devuelva los trienios que hay entre dos fechas (un trienio son tres años).
- 8 Codifica un procedimiento que reciba una lista de hasta cinco números y visualice su suma.
- 9 Escribe una función que devuelva solamente caracteres alfabéticos sustituyendo cualquier otro carácter por blancos a partir de una cadena que se pasará en la llamada.
- 10 Codifica un procedimiento que permita borrar un empleado cuyo número se pasará en la llamada.
- 11 Escribe un procedimiento que modifique la localidad de un departamento. El procedimiento recibirá como parámetros el número del departamento y la nueva localidad.
- 12 Visualiza todos los procedimientos y funciones del usuario almacenados en la base de datos y su situación (*valid* o *invalid*).