Capítulo 1

El lenguaje PL/SQL

En este capítulo se utilizará la base de datos que sirve para gestionar los equipos de hockey de varias ciudades, así como sus jugadores. El modelo E-R para esta base de datos se muestra en la Figura 1.1.

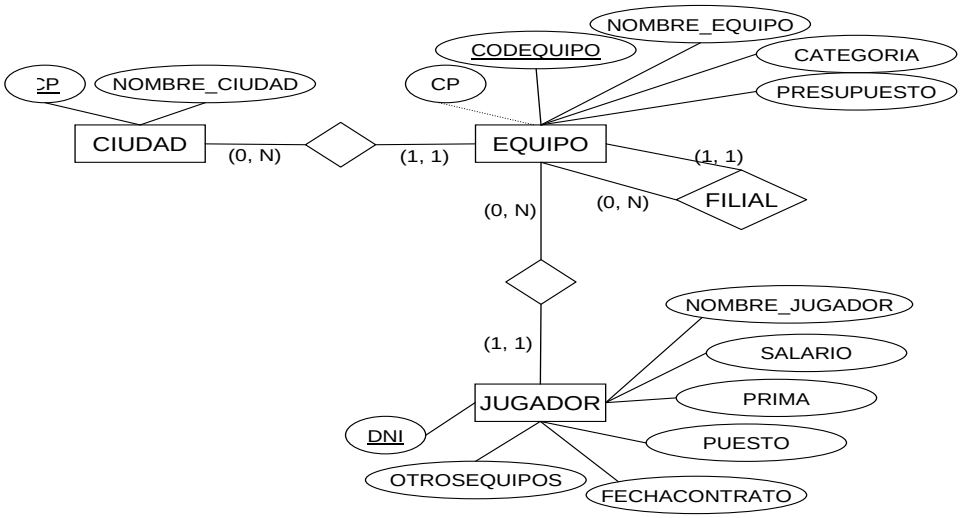

Figura 1.1: Modelo E-R de equipos de hockey

Una vez traducido al modelo relacional, e implementado en una base de datos, la sentencia de creación de las tablas sería la siguiente (además, se muestra, entre comentarios, la descripción de los campos):

```
CREATE TABLE CIUDAD(
   CP NUMBER(5) NOT NULL,        -- Código postal principal de la ciudad
   NOMBRE_CIUDAD VARCHAR2(15),   -- Nombre de la ciudad
   CONSTRAINT PK_CIUDAD PRIMARY KEY(CP)
   );

CREATE TABLE EQUIPO(
   CODEQUIPO CHAR(10) NOT NULL,  -- Código del equipo
   NOMBRE_EQUIPO VARCHAR2(20),   -- Nombre del equipo
   CATEGORIA VARCHAR2(20),       -- Categoría en la que juega
```

```
   PRESUPUESTO NUMBER(11,2),     -- Presupuesto del equipo, en euros
   EQPRINCIPAL CHAR(10),         -- Código del equipo principal, si es filial
   CP NUMBER(5),                 -- Código postal de la ciudad sede del equipo
   CONSTRAINT PK_EQUIPO PRIMARY KEY(CODEQUIPO),
   CONSTRAINT FK_CIUDAD FOREIGN KEY(CP) REFERENCES CIUDAD(CP),
   CONSTRAINT FK_EQPRIN FOREIGN KEY(EQPRINCIPAL) REFERENCES EQUIPO(CODEQUIPO)
   );

CREATE TABLE JUGADOR(
   DNI CHAR(11) NOT NULL,        -- DNI del jugador
   NOMBRE_JUGADOR VARCHAR2(15),  -- Nombre del jugador
   SALARIO NUMBER(7,2),          -- Salario anual del jugador, en euros
   PRIMA NUMBER(7,2),            -- Prima anual fija del jugador, en euros
   PUESTO VARCHAR2(10),          -- Puesto del jugador
   FECHACONTRATO DATE,           -- Fecha de contrato
   OTROSEQUIPOS NUMBER(2),       -- Número de (otros) equipos en que ha jugado
   CODEQUIPO CHAR(10),           -- Código del equipo en el que juega
   CONSTRAINT PK_JUGADOR PRIMARY KEY(DNI),
   CONSTRAINT FK_EQUIPO FOREIGN KEY(CODEQUIPO) REFERENCES EQUIPO(CODEQUIPO),
   CONSTRAINT C_PRIMAPOS CHECK (PRIMA >=0),
   CONSTRAINT C_SALPOS CHECK (SALARIO >=0)
   );
```

Para los ejemplos que se utilizarán en este tema, las tablas tienen la siguiente extensión:

```
  -- CIUDAD --
       CP NOMBRE_CIUDAD
---------- ---------------
     15000 A CORUÑA
     28000 MADRID
     27000 LUGO

3 filas seleccionadas.

  -- EQUIPO --
CODEQUIPO NOMBRE_EQUIPO CATEGORIA           PRESUP EQPRINCIPA CP
--------- ------------ ------------------ -------- -------- ----------
HCL         H.C. LICEO  DIVISION DE HONOR   1500000 <NULO>  15000
LB          LICEO B     PRIMERA DIVISION     900000  HCL     15000
ALC         ALCOBENDAS  DIVISION DE HONOR   2000000 <NULO>  28000
CHM         C.H. MADRID DIVISION DE HONOR   3100000 <NULO>  28000

4 filas seleccionadas.

  -- JUGADOR --
DNI     NOMBRE_JUGADOR  SALARIO PRIMA   PUESTO    FECHA  OTROS COD
------  --------------- ------- ------ ---------- ------- ---- -----
12345678  PEREZ, PEDRO  20000   <NULO>  PORTERO   01/06/01 1 HCL
12335678  PEREZ, PABLO  23000   3000    DELANTERO 01/09/01 2 HCL
22552278  ALVAREZ, JUAN 18000   0       MEDIO     14/02/02 0 HCL
33333334  ZAS, ANTONIO  17500   0       MEDIO     05/11/01 0 HCL
28342228  PARRA, JOSE   10000   <NULO>  PORTERO   21/01/02   LB
28321321  CASTOR, LUIS  9000    2000    DELANTERO 01/06/02 2 LB
24555555  CASA,FERNANDO 95000   <NULO>  MEDIO     23/04/02 0 LB
```

```
22178678  ARAUJO, JUAN  40000   0       PORTERO   11/02/02 1 ALC
22375989  ARAS, MIGUEL  23000   0       DELANTERO 21/09/01 2 ALC
22511111  ALVAREZ,ANGEL 19000   0       MEDIO     14/02/02 0 ALC

10 filas seleccionadas.
```

1.1 Introducción

Oracle es un gestor de bases de datos relacional, y utiliza para acceder a las bases de datos el lenguaje SQL. SQL es un lenguaje de cuarta generación, lo que se suele interpretar diciendo que el lenguaje se usa para describir lo que debe hacerse, pero no la manera de llevarlo a cabo. Sin embargo, esta potencia y facilidad de uso tiene sus inconvenientes: nos permite insertar datos, pero deben ser tuplas completas (con posibles valores nulos), y borrar o modificar datos sobre tablas, basándose en condiciones más o menos complejas que dependen de los datos existentes, pero actuando siempre a nivel de filas. También podemos recuperar datos, pero estos se devuelven siempre como una nueva relación. Así, supongamos que (considerando los datos de las tablas `JUGADOR` y `EQUIPO`) deseamos decrementar un 2% el salario del jugador que más cobra de cada equipo, pero sólo si aún con esa bajada sigue siendo el salario más alto del equipo. Además, esa actualización no puede hacer que el salario medio del equipo sea menor que la media de salarios de cualquiera de los equipos filiales del mismo. Esta actualización probablemente no sea posible en una única sentencia `UPDATE`, dada su complejidad, por lo que necesitaremos hacerlo utilizando otro lenguaje.

Existe otro tipo de lenguajes de programación, de tercera generación, como C y COBOL, que son de naturaleza más procedimental. Esto implica que pueden ser más complicados, pero también que ganan en potencia. Así, estos disponen de opciones no disponibles en el SQL, como variables, estructuras de control condicionales (`IF-THEN-ELSE`), bucles (`FOR`, etc.), etc. En un programa escrito en estos lenguajes, los algoritmos se implementan paso a paso para resolver el problema.

PL/SQL (Procedural Language/SQL) es un lenguaje que combina la potencia de los lenguajes de tercera generación y del SQL, ya que permite usar directamente SQL (veremos alguna restricción respecto a esto) pero también el uso de un lenguaje procedimental. PL/SQL está integrado en el servidor, por lo que suele ser bastante eficiente. Normalmente el código del PL/SQL es interpretado, aunque puede ser también compilado.

Se trata de un lenguaje particular de Oracle, que puede usarse para programar un servidor de BD y aplicaciones asociadas. Tiene sentencias que permiten declarar variables y constantes, controlar el flujo del programa, asignar y manipular datos y mucho más. En este capítulo veremos sus características básicas y algunos ejemplos de uso.

1.2 Características principales de PL/SQL

La unidad básica en PL/SQL es el **bloque**. Todos los programas PL/SQL están compuestos por bloques, que pueden estar situados de manera secuencial o anidados. Normalmente, cada bloque realiza una unidad lógica de trabajo en el programa, separando así unas tareas de otras.

Hay muchos tipos diferentes de bloques:

- **bloques anónimos**: se construyen, por regla general, de manera dinámica y se ejecutan una sola vez.
- **bloques nominados**: son bloques anónimos con una etiqueta que le da al bloque un nombre. También se construyen, generalmente, de manera dinámica y se ejecutan una sola vez.
- **subprogramas**: son procedimientos, paquetes y funciones almacenados en la base de datos. Estos bloques no cambian, por regla general, después de su construcción y se ejecutan múltiples veces. Los subprogramas se ejecutan explícitamente mediante una llamada al procedimiento, paquete o función.

- **disparadores**: son bloques nominados que también se almacenan en la base de datos. Tampoco cambian, generalmente, después de su construcción y se ejecutan múltiples veces. Los disparadores se ejecutan de manera implícita cada vez que tiene lugar el suceso de disparo. El suceso de disparo es una orden DML que se ejecuta sobre una tabla de la base de datos.

Un bloque tiene la siguiente estructura:

```
[DECLARE]
/* Sección declarativa - variables, tipos, cursores y subprogramas locales */
BEGIN
/* Sección ejecutable - órdenes PL/SQL */
[EXCEPTION]
/* Sección de manejo de excepciones */
END;
```

La única obligatoria es la sección ejecutable, y debe contener al menos una orden. La sección declarativa es opcional, sólo se incluirá si se declaran variables, o se definen cursores (veremos más adelante y en profundidad el concepto de cursor), constantes, tipos o subprogramas locales. La sección de manejo de excepciones sólo se utilizará cuando se quieran controlar explícitamente las excepciones o errores que la ejecución del código pueda generar. Utilizando esta sección se separa el código de gestión de errores del cuerpo principal del programa, con lo que se consigue que la estructura de éste sea más clara.

Así, el siguiente ejemplo nos permitiría ejecutar un bloque de código mínimo. Nótese que la barra (`/`) después del `END` no forma parte del bloque, sólo es para ejecutarlo.

```
BEGIN
   NULL;  -- Al menos una sentencia es obligatoria. NULL es una sentencia que no hace nada
END;
/
```

Dado que vamos a probar la ejecución de bloques de código, si hacemos que Oracle escriba algo en la pantalla, debe verse esa salida. Esto puede ser controlado en SQL*Plus con el parámetro `SERVEROUTPUT`. Para verlo usaremos `SHOW SERVEROUTPUT`, y lo activaremos con la orden `SET SERVEROUTPUT ON` que permite habilitar el valor de la variable SERVEROUTPUT de configuración de entorno de SQL*Plus.

Los procedimientos usados para mostrar mensajes por pantalla están en el paquete `DBMS_OUTPUT`, y básicamente usaremos las siguientes:

- `DBMS_OUTPUT.PUT_LINE(<argumento>)`: Escribe el `<argumento>` en la pantalla, y acaba con un retorno de carro. Es similar a `println` de C o al método `System.out.println` de Java.
- `DBMS_OUTPUT.PUT(<argumento>)`: Pone en el buffer de salida el `<argumento>`, pero no lo escribe hasta recibir la orden `DBMS_OUTPUT.NEW_LINE`. Se suele utilizar para escribir varias cosas en una línea.

Veamos un ejemplo:

```
SET SERVEROUTPUT ON
BEGIN
   DBMS_OUTPUT.PUT_LINE('Estamos a ' || SYSDATE);
   DBMS_OUTPUT.PUT('a ');
   DBMS_OUTPUT.PUT('b');
   DBMS_OUTPUT.NEW_LINE;
END;
/
Estamos a 13-NOV-02
a b
```

1.3 Unidades léxicas

1.3.1 Uso de etiquetas

PL/SQL permite utilizar etiquetas o *labels* que identifican un punto en el código. Esto sirve para utilizar las sentencias `GOTO` que hacen que el programa ejecute la siguiente línea a la etiqueta. Este estilo de programación es altamente desaconsejable, porque hace que el código sea muy difícil de entender, por lo que aquí no usaremos las etiquetas para esto.

Sin embargo, sí usaremos las etiquetas para identificar partes del código, como por ejemplo bucles. Una etiqueta se especifica en PL/SQL mediante `<<nombre_etiqueta>>`, donde los símbolos `<<` y `>>` también deben escribirse. Por ejemplo,

```
BEGIN
  /* Código */
  <<etiqueta_1>>
  /* Más código */
```

1.3.2 Tipos de literales

Un **literal** es un valor numérico, booleano o de carácter que no es un identificador. Dos ejemplos de literales serían: -23456 y NULL.

Los **literales de carácter**, también conocidos como literales de cadena, constan de uno o más caracteres delimitados por comillas simples. Pueden asignarse a variables de tipo `CHAR` o `VARCHAR2`, sin que sea necesario efectuar ningún tipo de conversión. Ejemplos válidos serían: '12345', 'Four score and seven years ago ...', '100%' ó ''''.

Todos los literales de cadena se consideran como de tipo `CHAR`. En un literal se puede incluir cualquier carácter imprimible del conjunto de caracteres de PL/SQL, incluyendo el propio carácter de comilla simple.

Un **literal numérico** representa un valor entero o real y puede ser asignado a una variable de tipo `NUMBER` sin necesidad de efectuar conversión alguna. Son los únicos literales que pueden emplearse como parte de una expresión aritmética. Los literales enteros consisten en una serie de dígitos, precedidos opcionalmente por un signo (+ ó -). No se permite utilizar un punto decimal en un literal entero. Literales enteros válidos son: 123, -7, +12, 0.

Un literal real consta de un signo, opcional, y una serie de dígitos que contiene un punto decimal, como en los siguientes ejemplos: -17.1, 23.0, 3. También se puede utilizar notación científica.

Por último, existen tres posibles **literales booleanos**: `TRUE`, `FALSE` y `NULL`. Estos valores sólo pueden asignarse a una variable booleana. Representan la verdad o falsedad de una condición y se utilizan en las órdenes `IF` y `LOOP`.

1.3.3 Comentarios

Los comentarios tienen por objeto facilitar la lectura y permiten entender mejor el código fuente de los programas. El compilador PL/SQL ignora los comentarios al compilar el programa. Existen dos clases de comentarios:

- Comentarios monolínea: Comienza con dos guiones y continúa hasta el final de la línea.
- Comentarios multilínea: Empiezan con el delimitador /* y terminan con el delimitador */. Este es el estilo utilizado por el lenguaje C. Los comentarios multilínea pueden ocupar tantas líneas como se desee. Sin embargo, no se puede anidar unos comentarios dentro de otros. Es necesario terminar un comentario antes de que comience otro.

1.4 Declaraciones de variables y tipos

La información se transmite entre PL/SQL y la base de datos mediante variables, que deben ser declaradas en la sección declarativa del bloque, después de la palabra clave `DECLARE`, y una sola variable por línea, en la que se indica la variable y su tipo. Cada declaración de variable termina en un punto y coma (;).

La sintaxis completa de declaración de una variable es la siguiente:

```
<nombre_variable> [CONSTANT] <tipo> [NOT NULL] [{:=|DEFAULT}<valor
por defecto>];
```

Los identificadores o nombres de las variables siguen las mismas normas que los nombres de las columnas, es decir, deben empezar por una letra (no se distingue entre mayúsculas y minúsculas), y contener sólo letras (del alfabeto inglés), números, o los símbolos $, _, ó # (y no ser una palabra reservada de PL/SQL). Si se quiere diferenciar, en el nombre de identificador, entre mayúsculas y minúsculas, incluir caracteres como los de espaciado o utilizar una palabra reservada, es necesario encerrar el identificador entre comillas dobles (").

La palabra `CONSTANT` indica que la "variable" es realmente una constante, por lo que debe ser inicializada en el momento de su definición, y su valor no puede ser modificado más tarde. Si se indica `NOT NULL`, la variable no admite valores nulos, por lo que debe ser también inicializada en el momento de su definición. La inicialización puede hacerse utilizando indistintamente el operador `:=` o `DEFAULT` seguido del valor con el que se inicializa, que debe ser una constante del mismo tipo que el especificado para la variable.

1.4.1 Tipos PL/SQL

Existen cuatro categorías de tipos: escalar, compuesto, referencias y LOB. Los tipos escalares no tienen componentes mientras que los compuestos sí los tienen. Las referencias son punteros a otros tipos. Los tipos PL/SQL se definen en un paquete denominado `STANDARD`, cuyos contenidos son accesibles desde cualquier bloque PL/SQL. Además de los tipos, este paquete define las funciones predefinidas SQL y de conversión en PL/SQL.

Tipos escalares

Son los mismos tipos que pueden usarse en una columna de base de datos más algunos tipos adicionales:

1. Tipos numéricos: Almacenan valores enteros o reales. Incluye todos los tipos numéricos propios de Oracle: NUMBER, DECIMAL, DOUBLE PRECISION, INTEGER, BINARY_INTEGER, NUMERIC, REAL y SMALLINT.

2. Tipos de carácter: Las variables de tipo carácter permiten almacenar cadenas o datos de tipo carácter. Esta familia de tipos está compuesta por VARCHAR2 y CHAR, entre otros.

   La sintaxis de VARCHAR2 es VARCHAR2(L), donde L es la longitud máxima de la variable. Es imprescindible indicar la longitud máxima, no existiendo un valor predeterminado. La longitud máxima se especifica en bytes, no en caracteres.

   La sintaxis de CHAR es CHAR(L). Las cadenas de caracteres de este tipo son de longitud fija.

3. Tipos raw: Se emplean para almacenar datos binarios. Las variables RAW son similares a las variables CHAR, excepto en que no se realizan conversiones entre conjuntos de caracteres. La sintaxis es RAW(L), donde L es la longitud en bytes de la variable. Se emplea para almacenar datos binarios de longitud fija.

4. Tipos de datos de fecha: Esta familia tiene un único miembro: DATE. Este tipo se comporta de la misma manera que el tipo equivalente de la base de datos. Se emplea para almacenar información sobre la fecha y la hora.

5. Tipo Rowid: El único tipo de esta familia es ROWID. Es el mismo que el tipo de pseudocolumna ROWID de la base de datos. En él se puede almacenar un identificador de columna, que es una clave que identifica unívocamente a cada fila de la base de datos.

6. Tipos booleanos: El único tipo de la familia es BOOLEAN. Las variables booleanas se emplean en las estructuras de control como IF-THEN-ELSE o LOOP. Puede contener los valores TRUE, FALSE o NULL.

A continuación se muestran algunos ejemplos válidos de declaraciones de variables:

```
DECLARE
  num1                NUMBER;                 -- var. numérica "normal"
  v_Nombre            VARCHAR2(20);           -- var. carácter de long. variable
  fecha1              DATE :=SYSDATE;         -- var. de tipo fecha
                                              -- inicializada a fecha actual
  v_BalanceCuenta     NUMBER(9,2);
  " nombre rarísimo"  NUMBER(1);
  v_ProcesoFinalizado BOOLEAN;
  num3                NUMBER DEFAULT 3;       -- default y := son equivalentes
  num4                CONSTANT NUMBER:=4;     -- constante, se debe inicializar
  num5                NUMBER NOT NULL DEFAULT 5; -- No admite nulos,
                                              -- hay que inicializarla
  num6                CONSTANT NUMBER NOT NULL := 6; -- constante no nula inicializada
```

Tipos compuestos

En PL/SQL hay disponibles tres tipos compuestos: registros, tablas y varrays. Un tipo compuesto es aquel que consta de una serie de componentes. Una variable de tipo compuesto contendrá una o más variables escalares.

Tipos de referencia

Una vez que se declara una variable de tipo escalar o compuesto se asigna el espacio de memoria de dicha variable. La variable se emplea para dar nombre a ese espacio de memoria y para hacer referencia a él en el programa. Sin embargo, no hay ninguna forma de desasignar este espacio de memoria y que la variable continúe estando disponible. La memoria no se libera hasta que no se sale del ámbito de la variable. Los tipos de referencia, que son el equivalente a un puntero en C, no tienen esta restricción. Cuando se declara una variable de un tipo de referencia, ésta puede apuntar a diferentes posiciones de memoria durante la vida del programa.

Tipos LOB

Se emplean para almacenar objetos de gran tamaño. Un objeto de gran tamaño es un valor binario o de tipo carácter con un tamaño de hasta 4 gigabytes. Los objetos de gran tamaño pueden contener datos no estructurados a los que se puede acceder de forma más eficiente y con menos restricciones que a los datos de tipo LONG o LONG RAW. Los tipos LOB se manipulan directamente mediante el paquete DBMS_LOB.

Tipos definidos por el usuario

PL/SQL también admite tipos definidos por el usuario, como tablas y registros. Los tipos definidos por el usuario permiten personalizar la estructura de los datos manipulados por un programa:

```
DECLARE
   TYPE t_RegistroEstudiante IS RECORD (
          Nombre    VARCHAR2(20),
          Apellido1 VARCHAR2(20),
          Apellido2 VARCHAR2(20),
          DNI       VARCHAR2(10)
         );
v_Estudiante t_RegistroEstudiante;
```

1.4.2 Uso de %TYPE y %ROWTYPE

En muchos casos, definiremos variables para acceder a datos almacenados en una tabla. En ese caso, definiremos las variables del mismo tipo que las columnas de la tabla. El "operador" `%TYPE` nos permite obtener el tipo de una variable o de una columna de una tabla. Así, si vamos a usar la variable `Sueldo` para acceder al salario de un jugador (columna `JUGADOR` en la tabla `JUGADOR`), deberemos declarar una variable del mismo tipo, es decir, `NUMBER(7,2)`. Sin embargo, si lo hacemos así y luego se modifica la columna, cambiando su tipo, habría que revisar todo el código para actualizar el tipo:

```
DECLARE
  Sueldo NUMBER(7,2);  --Puede dar lugar a problemas
                       --si se cambia la definición de JUGADOR.SALARIO
```

Una forma más correcta será indicar al sistema que esa variable es del mismo tipo que la columna correspondiente:

```
DECLARE
  Sueldo     JUGADOR.SALARIO%TYPE;  --Se indica el tipo de la columna correspondiente
  OtroSueldo Sueldo%TYPE;           --Podemos acceder al tipo de las variables
```

También es posible crear un registro (`RECORD`) que tenga la misma estructura que una tabla, usando `%ROWTYPE`. Así, la siguiente declaración sería correcta:

```
DECLARE
   regJugadores  JUGADOR%ROWTYPE; -- Ahora podemos acceder a regJugadores.DNI, etc.
```

El tipo se determina cada vez que se ejecuta el bloque, en el caso de bloques anónimos o nominados o cada vez que se compila algún objeto almacenado (procedimiento o función).

Si se aplica `%TYPE` a una variable o columna que haya sido definida con la restricción `NOT NULL`, el tipo devuelto no tiene esta restricción.

La utilización de `%TYPE` constituye una buena práctica de programación porque hace que los programas PL/SQL sean más flexibles y capaces de adaptarse a las definiciones cambiantes de la base de datos.

1.5 La sección ejecutable

La sección ejecutable de un bloque de código estará formada por sentencias y por estructuras de control, básicamente condicionales y bucles. Las sentencias son las que realmente realizan acciones, y pueden ser sentencias SQL, llamadas a procedimientos (como `DBMS_OUTPUT.PUT_LINE`) o funciones, o manipulación de variables. En cualquier caso, cada sentencia debe terminar con un punto y coma.

1.5.1 Manipulación de variables

Las variables pueden ser manipuladas directamente, asignándoles un valor o consultando el que tienen, o a través de sentencias SQL. El operador de asignación es `:=`, de la forma `<variable> := <valor>`. El valor es una expresión que debe ser obligatoriamente del mismo tipo que la variable a la que se asigna.

El siguiente código muestra cómo asignar valores a variables y mostrar por pantalla su contenido.

```
DECLARE
  Fecha DATE;
  Numero NUMBER(4);
BEGIN
  Fecha := SYSDATE;
  Numero := 45;
  Numero := Numero + 1; --Incrementa Numero en 1, ahora vale 46
  DBMS_OUTPUT.PUT_LINE('Estamos a '|| SYSDATE);
  DBMS_OUTPUT.PUT_LINE(Numero);
END;
```

El operador `||` permite la concatenación de cadenas de caracteres. Si todos los operandos de una expresión de concatenación son de tipo `CHAR`, entonces la expresión también lo es. Si alguno de los operandos es de tipo `VARCHAR2`, entonces la expresión también lo es. Los literales de cadena se consideran de tipo `CHAR`.

El acceso a datos almacenados se realiza en PL/SQL utilizando sentencias SQL. Podemos utilizar directamente sentencias DML (`INSERT`, `DELETE`, `UPDATE`) y también realizar consultas (`SELECT`), utilizando tanto constantes como variables para manipular u obtener datos de las tablas. No es posible, en cambio, ejecutar sentencias DDL del tipo `CREATE TABLE` directamente en PL/SQL, aunque sí hay una forma alternativa de ejecutar ese tipo de sentencias (utilizando SQL dinámico, pero no entraremos en más detalle sobre él aquí).

Las sentencias DML se escriben directamente en el código PL/SQL, y forman una sentencia "normal", pudiendo estar en un bucle o en una de las ramas de una estructura condicional. Cuando la sentencia necesite un valor, este puede ser una constante o una variable, previamente declarada, del mismo tipo que la columna que referencia.

Así, el siguiente bloque modifica los datos del jugador de DNI `12.345.678`, indicando que realmente ha estado en 3 equipos, y luego inserta un nuevo equipo, usando variables (se podrían usar también directamente las constantes):

```
DECLARE
  NumEquipos JUGADOR.OTROSEQUIPOS%TYPE;
  regEquipo EQUIPO%ROWTYPE;
BEGIN
  NumEquipos:=3;
  UPDATE JUGADOR
     SET OTROSEQUIPOS=NumEquipos
     WHERE DNI='12.345.678';
  regEquipo.CODEQUIPO:='INV';
  regEquipo.NOMBRE_EQUIPO:='Equipo Inventado';
  regEquipo.CATEGORIA:='No hay';
  regEquipo.PRESUPUESTO:=0;
  regEquipo.EQPRINCIPAL:=NULL;
  regEquipo.CP:=NULL;
  INSERT INTO EQUIPO
      VALUES regEquipo;
END;
```

Para asignar valores a variables a través de sentencias `SELECT` tendremos varios casos. En primer lugar, podemos usar la cláusula `INTO`, pero sólo si estamos completamente seguros de que la sentencia devuelve **exactamente una fila**:

```
DECLARE
  NumEquipos JUGADOR.OTROSEQUIPOS%TYPE;
  regEquipo EQUIPO%ROWTYPE;
BEGIN
  SELECT OTROSEQUIPOS
     INTO NumEquipos
     FROM JUGADOR
     WHERE DNI='12.345.678';

  SELECT *
     INTO regEquipo
     FROM EQUIPO
     WHERE CODEQUIPO='HCL';
END;
```

Obsérvese que en la segunda sentencia, la que obtiene datos del equipo, podemos devolver todos los datos del centro directamente al registro, lo que hace que cada campo del mismo obtenga el valor de la columna correspondiente. Es decir, `regEquipo.CODEQUIPO` recoge el valor de `EQUIPO.CODEQUIPO` devuelto por la consulta, y lo mismo sucede con las demás columnas.

Como se ha indicado, la cláusula `INTO` sólo se puede usar si la consulta devuelve exactamente una fila. Esto es lógico, ya que una variable sólo puede almacenar un valor, y si intentamos asignar más de uno da un error. En concreto, esta situación produce la excepción denominada `TOO_MANY_ROWS`, y que si no se controla explícitamente produce el siguiente resultado:

```
DECLARE
   regEquipo EQUIPO%ROWTYPE;
BEGIN
   SELECT *
       INTO regEquipo
       FROM EQUIPO;
END;
/
DECLARE
 *
ERROR en línea 1: ORA-01422: la recuperación exacta devuelve un
número mayor de filas que el solicitado ORA-06512: en línea 4
```

En el caso en que la consulta no devuelva ninguna fila, también se produce un error (en este caso, la excepción `NO_DATA_FOUND`), ya que no hay valores para asignar a las variables:

```
DECLARE
   regEquipo EQUIPO%ROWTYPE;
BEGIN
   SELECT *
     INTO regEquipo
     FROM EQUIPO
     WHERE CODEQUIPO='ABC';
END;

DECLARE * ERROR en línea 1: ORA-01403: no se han encontrado datos
ORA-06512: en línea 4
```

Veremos más adelante el concepto de *cursor*, que nos permite hacer consultas de las que no sabemos el número de filas que se obtienen. El manejo de excepciones como las que se presentan en los casos anteriores se explica en la Sección 1.6.

1.5.2 Estructuras de control

El control del flujo de un programa se realiza básicamente mediante tres tipos de estructuras de control: secuencia, condicional y bucle.

Una secuencia es una colección de sentencias que se ejecutan en determinado orden, y que se consigue escribiendo las sentencias una detrás de otra, como en los ejemplos anteriores. Una estructura condicional, normalmente implementada con estructuras `IF-THEN-ELSE`, evalúa una condición lógica, y en función de si es cierta o no elige un camino. Un bucle es un bloque de código que se ejecuta de forma repetitiva. Veamos a continuación cómo se implementan en Oracle las estructuras condicionales y los bucles:

Estructuras condicionales

La estructura condicional, también denominada alternativa, se basa en evaluar una condición lógica, y dependiendo del resultado de esa evaluación, se elige uno u otro camino de ejecución. La sintaxis completa de una estructura condicional es la siguiente:

```
IF <condición 1> THEN
/* secuencia de acciones a ejecutar si condición 1 es cierta */
[ELSIF <condición 2> THEN
/* secuencia de acciones a ejecutar si condición 2 es cierta
(y condición 1 es falsa o nula) */
...  ]
[ELSE /*secuencia de acciones a ejecutar si todas las condiciones son falsas o nulas*/ ]
END IF;
```

Una sentencia `IF` básica tendría la siguiente estructura:

```
IF <condición> THEN
   <secuencia que se ejecuta si condición es cierta>
END IF;
```

Por ejemplo, suponiendo una variable numérica `Numero`, podemos imprimirla si vale 3:

```
IF Numero = 3 THEN
   DBMS_OUTPUT.PUT_LINE('La variable Numero vale 3');
END IF;
```

Opcionalmente, podemos completar la sentencia if con la parte `ELSE` que se ejecutará si la condición no es cierta:

```
IF Numero = 3 THEN
   DBMS_OUTPUT.PUT_LINE('La variable Numero vale 3');
ELSE
   DBMS_OUTPUT.PUT_LINE('La variable Numero NO vale 3');
END IF;
```

La parte `ELSIF` (de hecho, puede haber tantos como se deseen) sirve como una abreviatura de un `ELSE IF...`, es decir, poner una nueva estructura `IF` dentro de la parte `ELSE`. Por ejemplo, el siguiente bloque de código decide si un número es menor que 5, está entre 5 y 10, o es mayor que 10:

```
DECLARE
   Numero NUMBER;
BEGIN
   /*  ... La variable Numero toma un valor ... */
   IF Numero < 5 THEN
          -- Si (Numero<5) es cierto...
      DBMS_OUTPUT.PUT_LINE('Es menor que 5');
   ELSIF Numero <= 10 THEN
          -- Si (Numero < 5) no es cierto...
          -- y (Numero <= 10) sí lo es...
      DBMS_OUTPUT.PUT_LINE('Está entre 5 y 10');
   ELSE
          -- Si todas las condiciones anteriores son falsas...
      DBMS_OUTPUT.PUT_LINE('Es mayor que 10');
   END IF;
END;
```

Estructuras de bucle

Un bucle es una estructura repetitiva, que nos permite ejecutar varias veces una secuencia de órdenes. El bucle básico en Oracle se implementa con la estructura `LOOP ... END LOOP;`. Veamos un ejemplo que muestra secuencialmente los números comenzando en 1:

```
 DECLARE
    contador BINARY_INTEGER := 1;
 BEGIN
    LOOP                                    -- Bucle infinito!!!
        DBMS_OUTPUT.PUT_LINE(contador);
        contador := contador + 1;
    END LOOP;
 END;
```

Este bucle muestra los números 1, 2, 3, ... por pantalla, ya que la variable está inicializada a 1 en la declaración, y se incrementa una unidad en cada paso del bucle. Sin embargo, es un bucle infinito, ya que no hay una sentencia que indique que se debe salir del bucle, Para ello se usa la sentencia `EXIT`. Así, si queremos imprimir los números del 1 al 10, podríamos hacer lo siguiente:

```
DECLARE
    contador BINARY_INTEGER := 1;
BEGIN
   LOOP
     DBMS_OUTPUT.PUT_LINE(contador);
     contador := contador + 1;
     IF contador > 10 THEN
        EXIT;
     END IF;
   END LOOP;
END;
```

La forma más correcta de salir (aunque equivalente a la anterior) es usar la sentencia

```
EXIT [<etiqueta>] WHEN condicion
```

Además, sobre todo si tenemos bucles anidados, debemos indicar de qué bucle queremos salir. Para ello usaremos las etiquetas definidas en el apartado 1.3.1. Un ejemplo es el siguiente:

```
DECLARE
   cont1 BINARY_INTEGER := 1;
   cont2 BINARY_INTEGER := 1;
BEGIN
   <<externo>>
   LOOP
      DBMS_OUTPUT.PUT_LINE('Bucle externo: '||cont1);
      cont2:=1;
      <<interno>>
      LOOP
         DBMS_OUTPUT.PUT(' Int:'||cont2);
         cont2:=cont2+1;
         EXIT interno WHEN cont2>cont1;
      END LOOP;
      DBMS_OUTPUT.NEW_LINE;
      cont1 := cont1 + 1;
      EXIT externo WHEN cont1 > 10;
   END LOOP;
END;
/
Bucle externo: 1
Int:1
Bucle externo: 2
Int:1 Int:2
Bucle externo: 3
Int:1 Int:2 Int:3
Bucle externo: 4
Int:1 Int:2 Int:3Int:4
Bucle externo: 5
Int:1 Int:2 Int:3 Int:4 Int:5
Bucle externo: 6
Int:1 Int:2 Int:3 Int:4 Int:5 Int:6
Bucle externo: 7
Int:1 Int:2 Int:3 Int:4 Int:5 Int:6 Int:7
Bucle externo: 8
Int:1 Int:2 Int:3 Int:4 Int:5 Int:6 Int:7 Int:8
Bucle externo: 9
Int:1 Int:2 Int:3 Int:4 Int:5 Int:6 Int:7 Int:8 Int:9
Bucle externo: 10
Int:1 Int:2 Int:3 Int:4 Int:5 Int:6 Int:7 Int:8 Int:9 Int:10

Procedimiento PL/SQL terminado correctamente.
```

Existe un tipo especial de bucle, el bucle `WHILE`, en el que se incluye directamente la condición que se debe verificar para seguir en el bucle. La estructura general es `WHILE<condicion> LOOP ... END LOOP`:

```
DECLARE
    contador BINARY_INTEGER := 1;
BEGIN
   WHILE contador <= 10 LOOP
        DBMS_OUTPUT.PUT_LINE(contador);
        contador := contador + 1;
   END LOOP;
END;
```

Sobre este tipo de bucles debemos indicar que la condición se comprueba siempre *antes* de ejecutar las sentencias del bucle, por lo que si la primera vez no se cumple, ya no entra en el bucle. En los anteriores, las sentencias se ejecutaban al menos una vez.

Sin embargo, para el tipo de ejemplos que estamos usando, podríamos usar el bucle numérico `FOR`, que tiene una sintaxis más simple:

```
FOR <variable> IN [REVERSE] <limite_inferior> ..<limite_superior> LOOP
/* Sentencias */
END LOOP;
```

Por ejemplo:

```
BEGIN
   FOR contador IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(contador);
   END LOOP;
END;
```

Este tipo de bucles tiene la particularidad de que la variable del bucle, `contador` en el ejemplo anterior, es declarada implícitamente, es decir, no hay que declararla en la sección de declaración del bloque PL/SQL.

El bucle inicializa la variable contador a `<limite_inferior>`, en cada pasada del bucle lo incrementa en uno y sale cuando esta variable se pasa del `<límite_inferior>` (en el ejemplo anterior, cuando vale 10 también se ejecuta). Si se usa `IN REVERSE`, la variable se inicializa a `<limite_superior>`, se va decrementando, y se sale cuando después de alcanzar `<limite_inferior>` . Nótese que siempre se indica primero el límite inferior y luego el superior, independientemente de si se usa `REVERSE` o no.

1.5.3 Consultas y cursores

Las consultas de datos, aunque todas se realizan enviando sentencias `SELECT` a la base de datos, tienen dos variantes: La primera, cuando se conoce con seguridad y a priori que la consulta va a devolver exactamente una fila, y la segunda, cuando no se sabe el número de filas que va a devolver, pudiendo ser ninguna, una o más.

Consultas que devuelven una fila

La primera variante, una única fila, suele utilizarse cuando se seleccionan datos que proceden de funciones de agregación, como `COUNT`, `SUM`, etc., o cuando se busca una fila a partir de su clave primaria, y se sabe que esa clave existe. En estos casos podemos utilizar la cláusula `INTO` de la sentencia `SELECT`, y obtener los datos, como ya se ha visto. La cláusula tiene dos posibles formatos:

```
SELECT <lista_de_atributos> INTO <lista_de_variables>;
ó
SELECT <lista_de_atributos> INTO <registro>;
```

Por ejemplo, para conocer información sobre el número total de jugadores existentes y la suma de sus salarios, por un lado, e información sobre el equipo cuyo código es HCL podríamos hacerlo así:

```
DECLARE
    CantidadDeJugadores NUMBER(2);
    SumaDeSalario       NUMBER(8,2);
    regEquipo           EQUIPO%ROWTYPE;
BEGIN
    SELECT COUNT(*), SUM(SALARIO)
    INTO CantidadDeJugadores, SumaDeSalario
    FROM JUGADOR;
    DBMS_OUTPUT.PUT_LINE('Hay '|| CantidadDeJugadores);
    DBMS_OUTPUT.PUT_LINE('En total cobran '|| SumaDeSalario);
    SELECT *
      INTO regEquipo
      FROM EQUIPO
      WHERE CODEQUIPO='HCL';
    DBMS_OUTPUT.PUT_LINE('Código: '||regEquipo.CODEQUIPO||
                         ' Nombre: '||regEquipo.NOMBRE_EQUIPO);
    DBMS_OUTPUT.PUT_LINE('Categoría: '||regEquipo.CATEGORIA||
                         ' Presupuesto:'||regEquipo.PRESUPUESTO);
    DBMS_OUTPUT.PUT_LINE('Equipo Principal:'||regEquipo.EQPRINCIPAL||
                         ' CP: '||regEquipo.CP);
END; / Hay 10 En total cobran 274500 Código: HCL       Nombre:
H.C. LICEO Categoría: DIVISION DE HONOR Presupuesto: 1500000
Equipo Principal:  CP: 15000

Procedimiento PL/SQL terminado correctamente.
```

Cursores

Un cursor es una estructura que se emplea para procesar múltiples (o ninguna, sin que se produzca la excepción `NO_DATA_FOUND`) filas extraídas de la base de datos.

Para poder procesar cualquier orden SQL, Oracle asigna un área de memoria que recibe el nombre de *área de contexto* o *área de SQL*. Esta área contiene información necesaria para completar el procesamiento, incluyendo el número de filas procesadas por la orden, un puntero a la versión analizada de la orden ejecutada y, en el caso de las consultas, el conjunto activo, que es el conjunto de filas resultado de la consulta. Un cursor es un puntero al área de SQL. Mediante el cursor, un programa PL/SQL puede controlar el área de SQL y lo que en ella sucede a medida que se procesa la orden.

Así, en el caso de las consultas, se usará un cursor para que el programa pueda procesar, una por una, las filas que la consulta devuelve. La forma general de manejar un cursor en PL/SQL es la siguiente:

1. **Declarar el cursor**: En la sección de declaración se asocia un nombre de cursor a una sentencia `SELECT`. Una declaración de cursor puede hacer referencia a variables PL/SQL en su cláusula `WHERE`. Estas variables deben ser visibles en el punto donde se declara el cursor. Para asegurarse de que todas las variables referenciadas en una declaración de cursor son declaradas antes de la referencia pueden declararse todos los cursores al final de la sección declarativa.

   Es importante indicar que sólo se está declarando, y no ejecutando, la consulta, por lo que no es necesario que las variables estén inicializadas.

   La sintaxis de la declaración de un cursor es

   ```
   CURSOR <nombre_cursor> IS <sentencia select>;
   ```

2. **Abrir el cursor**: Se ejecuta la consulta, y en una zona de memoria reservada para el cursor se almacenan las filas que esta devuelve. Si la consulta incluía variables (que se denominan variables *acopladas*), estas deben tener el valor deseado antes de abrir el cursor. Al abrir un cursor y ejecutar la consulta, un apuntador se coloca en la primera fila que se va a devolver. Una vez abierto el cursor, podemos cambiar el valor de las variables acopladas ya que esto no afecta al cursor. Esto es debido a que el conjunto activo, es decir, el conjunto de filas que satisfacen los criterios de la consulta se determina únicamente en el momento de abrir el cursor.

   Es legal abrir un cursor que ya está abierto. PL/SQL ejecutará implícitamente una orden CLOSE antes de reabrir el cursor con la segunda orden OPEN. También puede haber más de un cursor abierto al mismo tiempo.

   La sentencia para abrir un cursor es la siguiente:

   ```
   OPEN <nombre_cursor>;
   ```

3. **Procesar las filas** que devuelve el cursor. Este proceso es repetitivo, para procesar todas las filas una a una, por lo que se implementará en un bucle hasta que no haya más filas. Cada fila que se obtenga se almacena, al igual que se hacía con `SELECT ... INTO`, en una lista de variables o en un registro PL/SQL. La orden que obtiene la fila actual, y desplaza el apuntador del cursor a la siguiente fila, es `FETCH .. INTO`:

   ```
   FETCH <nombre_cursor> INTO lista de variables | registro PL/SQL;
   ```

   El cursor tiene una serie de propiedades, una de las cuales nos indica cuando se han acabado las filas. Esta propiedad, cuyo `<nombre_cursor>%NOTFOUND`, se usará para determinar cuando se sale del bucle. La última orden `FETCH`, cuando `%NOTFOUND` es cierto, no realizará una asignación de valores a las variables de salida, sino que éstas contendrán todavía los valores anteriores.

4. **Cerrar el cursor**: Cuando ya se han procesado todas las filas, se deben liberar los recursos (memoria, etc.) asociados al cursor. La sentencia para cerrar un cursor es:

   ```
   CLOSE <nombre_cursor>;
   ```

Evidentemente, una vez que se cierra un cursor es ilegal realizar extracciones (FETCH) de él, o tratar de volver a cerrarlo. Ambos casos provocarían un error.

Veamos un ejemplo, en el que se obtienen el nombre, salario y número de equipos en los que han estado los jugadores del equipo HCL:

```
DECLARE
    Nombre        JUGADOR.NOMBRE_JUGADOR%TYPE;
    Salario       JUGADOR.SALARIO%TYPE;
    NumeroEquipos JUGADOR.OTROSEQUIPOS%TYPE;
    CodigoEquipo  JUGADOR.CODEQUIPO%TYPE;
    --Declaramos el cursor
    CURSOR C_JUG IS
          SELECT NOMBRE_JUGADOR, SALARIO, OTROSEQUIPOS
                 FROM JUGADOR
                 WHERE CODEQUIPO = CodigoEquipo;
BEGIN
     -- Asignamos el valor a la variable que indica el equipo
     -- y abrimos el cursor
     CodigoEquipo:='HCL';
     OPEN C_JUG;

     /* Si cambiásemos el valor de la variable acoplada
     (a CodigoEquipo:='LB', por ejemplo) no afecta al cursor
     */

     LOOP
       FETCH C_JUG INTO Nombre, Salario, NumeroEquipos;
       EXIT WHEN C_JUG%NOTFOUND;
           -- El proceso es simplemente imprimir los valores
       DBMS_OUTPUT.PUT_LINE(Nombre||' '||Salario||' '||NumeroEquipos);
     END LOOP;
        -- Al acabar, cerramos el cursor
     CLOSE C_JUG;
END;
/
PEREZ, PEDRO  20000 1
PEREZ, PABLO  23000 2
ALVAREZ, JUAN 18000 0
ZAS, ANTONIO  17500 0

Procedimiento PL/SQL terminado correctamente.
```

Los cursores tienen cuatro atributos. Para acceder a ellos en un bloque PL/SQL, se añade al nombre del cursor, de forma similar a %TYPE o %ROWTYPE.

- %FOUND: Es un atributo booleano que devuelve cierto si la última orden FETCH devolvió una fila, y falso en caso contrario. Si se trata de comprobar el valor de <nombre_cursor>%FOUND mientras el cursor no está abierto, se devuelve el error ORA-1001 (cursor no válido).
- %NOTFOUND: Es lo opuesto a %FOUND, devuelve cierto si la orden FETCH no devuelve una fila, y falso si la devuelve.

- `%ISOPEN`: También booleano, devuelve cierto si el cursor está abierto, y falso en caso contrario.
- `%ROWCOUNT`: Es un atributo numérico devuelve el número de filas extraídas por el cursor hasta el momento.

Cursores implícitos

Los cursores explícitos, los vistos hasta ahora, sirven para procesar sentencias `SELECT` que devuelven un número indeterminado de filas. Sin embargo, todas las órdenes SQL se ejecutan dentro de un área de SQL y tienen, por tanto, un cursor que apunta a dicha área. Este cursor se conoce con el nombre de cursor `SQL`. A diferencia de los cursores explícitos, en el programa no se abre ni cierra el cursor `SQL`, sino que el motor de PL/SQL lo abre de modo implícito, procesa la orden SQL en él contenida y cierra el cursor después.

El cursor implícito sirve para procesar las órdenes `INSERT`, `UPDATE`, `DELETE` y las órdenes `SELECT...INTO` de una sola fila. Puesto que es el motor PL/SQL quien abre y cierra el cursor, las órdenes `OPEN`, `FETCH` y `CLOSE` no son relevantes para este tipo de cursor. Sin embargo, sí que se pueden aplicar al cursor `SQL` los atributos de cursor antes mencionados.

Veamos un ejemplo: Supongamos los siguientes esquemas de tablas, para facturas y líneas de factura:

$$FACTURA(\underline{Codigo}, Fecha, Total)$$

$$LINEA\_FACTURA(\underline{Codigo}, \underline{Linea}, Precio)$$

Se desea que al insertar una línea de factura, se incremente el total de la factura con el precio de la nueva línea, y si es la primera línea de la factura, que se cree la correspondiente tupla en la tabla de facturas:

```
DECLARE
   CodigoFactura LINEA_FACTURA.CODIGO%TYPE;
   Fecha DATE :=SYSDATE;
   NumeroLinea LINEA_FACTURA.LINEA%TYPE;
   PrecioLinea LINEA_FACTURA.PRECIO%TYPE;
BEGIN
   -- En esta parte se asignarían valores a la variables
   CodigoFactura := ...
   NumeroLinea := ...
   PrecioLinea := ...

   INSERT INTO LINEA_FACTURA (CODIGO,LINEA,PRECIO)
      VALUES(CodigoFactura, NumeroLinea, PrecioLinea);

   UPDATE FACTURA
      SET TOTAL = TOTAL + PrecioLinea
      WHERE CODIGO = CodigoFactura;

   -- La variable SQL%NOTFOUND se activa si no se han
   -- procesado filas
   IF SQL%NOTFOUND THEN
      INSERT INTO FACTURA(CODIGO,FECHA,TOTAL)
         VALUES(CodigoFactura, Fecha, PrecioLinea);
   END IF;
END;
```

En primer lugar insertamos la línea de factura. Luego ejecutamos una sentencia que actualiza TODAS las facturas (será una como máximo) que tengan el código indicado. Si no hay ninguna, la variable `SQL%NOTFOUND` se activa (se pone a cierto). El `IF` comprueba si se ha modificado alguna factura, y si no es así (es decir, si `SQL%NOTFOUND` es cierto), ejecuta la sentencia de inserción.

Bucles sobre cursores
Como hemos visto, el procesamiento de las filas que devuelve un cursor se realiza mediante bucles. La forma que se ha indicado antes es la que sigue más directamente los cuatro pasos teóricos (declarar, abrir, usar y cerrar el cursor). Sin embargo, hay otros bucles, en muchos casos más simples, que se pueden usar con un cursor. Éstos son los bucles `WHILE` y `FOR`.

Bucles de cursor `WHILE`: El siguiente es un ejemplo que recorre un cursor sobre todos los jugadores y muestra su nombre y salario, usando un bucle `WHILE`.

```
DECLARE
   Nombre        JUGADOR.NOMBRE_JUGADOR%TYPE;
   Salario       JUGADOR.SALARIO%TYPE;

   CURSOR C_JUG IS
         SELECT NOMBRE_JUGADOR, SALARIO
                  FROM JUGADOR;
BEGIN
   OPEN C_JUG;
   FETCH C_JUG INTO Nombre, Salario;
   WHILE C_JUG%FOUND LOOP
       DBMS_OUTPUT.PUT_LINE(Nombre||' cobra '||Salario);
       FETCH C_JUG INTO Nombre, Salario;
   END LOOP;
   CLOSE C_JUG;
END;
```

Este tipo de bucles utiliza lo que se denomina lectura adelantada, ya que se lee antes de iniciar el bucle (así se puede comprobar si hay o no datos) y luego, en el bucle, se procesa la fila leída y finalmente se obtiene la siguiente. Esto es necesario para que la condición del bucle sea evaluada en cada iteración del bucle.

Bucles de cursor `FOR`: Los dos tipos de bucles anteriores requieren un procesamiento explícito del cursor mediante las órdenes `OPEN`, `FETCH` y `CLOSE`. PL/SQL proporciona un nuevo tipo de bucle, mucho más simple, que realiza de modo implícito el procesamiento del cursor. Este bucle recibe el nombre de bucle de cursor `FOR`, y que tiene la sintaxis general siguiente:

```
FOR <registro PL/SQL> IN <nombre_cursor> LOOP
/* Procesar la fila actual */
END LOOP;
```

Veamos cómo quedaría el ejemplo anterior:

```
DECLARE
   CURSOR C_JUG IS
         SELECT NOMBRE_JUGADOR, SALARIO
                  FROM JUGADOR;
BEGIN
  FOR regJUGADOR IN C_JUG LOOP
    -- Procesamiento de filas...
    DBMS_OUTPUT.PUT_LINE(regJUGADOR.NOMBRE_JUGADOR || ' cobra '|| regJUGADOR.SALARIO);
  END LOOP;
END;
```

Este tipo de cursor tiene varias características especiales:

- El registro PL/SQL que se usa no se declara en la sección declarativa, sino que el bucle `FOR` lo declara de forma implícita, al igual que los índices de los bucles `FOR` numéricos. En este caso, además, el registro implícito tiene sólo los campos que se seleccionan (en el ejemplo, el nombre y el salario del jugador).
- No hay que abrir el cursor, ya que el `FOR` lo hace automáticamente antes del bucle, ni comprobar cuando se acaban las filas (el bucle comprueba automáticamente el atributo `%FOUND` antes de cada iteración), ni cerrarlo, ya que es cerrado automáticamente al terminar.

Como se ve, los bucles de cursor `FOR` tienen la ventaja de que proporcionan la funcionalidad de un bucle para recorrer un cursor de una forma simple y limpia con una sintaxis mínima.

Cursores actualizables

Los cursores vistos hasta ahora son de "sólo lectura": el cursor está asociado con una orden SQL y esta orden es conocida en el momento de compilar el bloque. Es decir, podemos obtener los datos, pero el conjunto activo que devuelve el cursor no es modificable.

En algún caso es útil poder actualizar la fila actual del cursor, y eso se puede implementar con *cursores actualizables*, o también llamados *cursores FOR UPDATE*. Estos cursores pueden asociarse con diferentes órdenes en tiempo de ejecución. Los cursores actualizables son análogos a las variables PL/SQL, que pueden contener valores diferentes en tiempo de ejecución. Los cursores estáticos (de sólo lectura), por su parte, son análogos a las constantes PL/SQL dado que sólo pueden ser asociados con una única consulta en tiempo de ejecución.

Para poder utilizar un cursor actualizable es necesario declararlo. Después será necesario asignarle espacio de almacenamiento en tiempo de ejecución. El resto de las operaciones (apertura, extracción de datos y cierre) son similares a las de los cursores explícitos.

Este tipo de cursores se declaran añadiendo `FOR UPDATE` después de la sentencia `SELECT`. Para actualizar la fila actual se usa una orden `UPDATE`, y la condición será `WHERE CURRENT OF <nombre_cursor>`, como se ve en el siguiente ejemplo.

Ejemplo: Usemos un cursor actualizable para aumentar el presupuesto en un 6%, sólo para aquellos equipos con menos de `3.000.000` (Los datos originales se muestran en el primer apartado).

```
DECLARE
    -- Se declara el cursor actualizable
    CURSOR C_EQUIPO IS
       SELECT *
          FROM EQUIPO
             WHERE PRESUPUESTO < 3000000
             FOR UPDATE;


BEGIN
    FOR regEquipo IN C_EQUIPO LOOP
      DBMS_OUTPUT.PUT_LINE(regEquipo.CODEQUIPO||' '||regEquipo.PRESUPUESTO);
      UPDATE EQUIPO SET PRESUPUESTO = PRESUPUESTO * 1.06
             WHERE CURRENT OF C_EQUIPO;
    END LOOP;
END;
/
HCL        1500000
LB         900000
ALC        2000000

Procedimiento PL/SQL terminado correctamente.
```

Véase que en la salida del bloque PL/SQL no se ven reflejados los cambios, porque se muestran los datos antes de la actualización. Los datos después de la ejecución de este código son los siguientes, en los que se ve que sí se han realizado cambios.

```
SQL> SELECT * FROM EQUIPO;

CODEQUIPO  NOMBRE_EQUIPO        CATEGORIA            PRESUPUESTO EQPRINCIPA         CP
---------- -------------------- -------------------- ----------- ---------- ----------
HCL        H.C. LICEO           DIVISION DE HONOR        1590000 <NULO>          15000
LB         LICEO B              PRIMERA DIVISION          954000 HCL             15000
ALC        ALCOBENDAS           DIVISION DE HONOR        2120000 <NULO>          28000
CHM        CLUB HOCKEY MADRID   DIVISION DE HONOR        3100000 <NULO>          28000
4 filas seleccionadas.
```

1.6 Manejo de excepciones

Una excepción es un error que se produce durante la ejecución de un bloque de PL/SQL. Si el error no se trata de forma explícita, Oracle lo gestiona automáticamente y muestra un código y un mensaje de error, fallando la ejecución del código. El siguiente ejemplo produce un error porque la sentencia `SELECT...INTO` no devuelve filas. Al no manejar la excepción explícitamente, Oracle gestiona el error.

```
DECLARE
    regEquipo EQUIPO%ROWTYPE;
BEGIN
    SELECT *
       INTO regEquipo
       FROM EQUIPO
       WHERE CODEQUIPO='121';
END;
/
DECLARE
 *
ERROR en línea 1: ORA-01403: no se han encontrado datos
ORA-06512:en línea 4
```

Oracle utiliza la variable numérica `SQLCODE` para indicar el código de error de sentencias SQL. Si la última sentencia ha sido exitosa, `SQLCODE` es 0 (cero), y en otro caso será un número negativo. La variable `SQLERRM` almacena el mensaje de error. Para el caso en el que no hay error, almacenan lo siguiente:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Código: '||SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Mensaje: '||SQLERRM);
END; / Código: 0 Mensaje: ORA-0000: normal, successful completion
```

Las excepciones se manejan en la sección de control de excepciones, que va situada después de la sección ejecutable del bloque y comienza por la palabra clave `EXCEPTION`. Esto nos permite separar el control de errores de la ejecución normal del código, con lo que se obtiene un código más claro y legible.

Las excepciones más comunes tienen un nombre en Oracle. Así, la anterior se llama `NO_DATA_FOUND`, y la que se produce cuando un `SELECT...INTO` devuelve más de una fila, `TOO_MANY_ROWS`. Para manejar una excepción concreta, se usa la sintaxis

```
WHEN <nombre_excepcion> THEN <sentencias de manejo de la excepción>;
```

Por ejemplo, para emitir un mensaje de error cuando el resultado de una consulta a la tabla EQUIPO devuelve más de una fila haremos:

```
DECLARE
  regEquipo EQUIPO%ROWTYPE;
BEGIN
  SELECT *
        INTO regEquipo
        FROM EQUIPO;
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Hay más de una fila');
END; / Hay más de una fila

Procedimiento PL/SQL terminado correctamente.
```

Así, se comprueba la excepción y se maneja de forma específica. Si se produce una excepción de otro tipo, la gestión es la que hace por defecto Oracle. Si queremos que todas las excepciones (excepto las indicadas específicamente) se gestionen de la misma forma, existe la palabra clave `OTHERS` que se refiere a las demás excepciones (o a todas si no se indica ninguna en una cláusula `WHEN` anterior).

Por ejemplo, para controlar la excepción producida al realizar una consulta de los equipos cuyo presupuesto sea 'ABC':

```
DECLARE
    regEquipo EQUIPO%ROWTYPE;
BEGIN
    SELECT *
       INTO regEquipo
       FROM EQUIPO
       WHERE PRESUPUESTO='ABC';
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
         DBMS_OUTPUT.PUT_LINE('Hay más de una fila');
    WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('Se ha producido un error');
END;

Se ha producido un error

Procedimiento PL/SQL terminado correctamente.
```

En este caso, la excepción que se trata de forma genérica es la siguiente, de número incorrecto (al tratar de comparar la columna `PRESUPUESTO` con la constante `'ABC'`): `ORA-01722: invalid number`.

Podríamos hacer una gestión genérica de la siguiente forma:

```
DECLARE
    regEquipo EQUIPO%ROWTYPE;
BEGIN
    SELECT *
       INTO regEquipo
       FROM EQUIPO
       WHERE PRESUPUESTO='ABC';
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Código: '||SQLCODE);
```

```
        DBMS_OUTPUT.PUT_LINE('Mensaje: '||SUBSTR(SQLERRM, 11,100));
    END;

Código: -1722 Mensaje:  número no válido
```

Además, podemos crear nuestras propias excepciones en la sección DECLARE, y manejarlas en el bloque de código. Para ello Oracle nos ofrece un tipo especial: `EXCEPTION`, y la posibilidad de provocar ("elevar") una excepción con la orden `RAISE`.

Por ejemplo, si se desea emitir un mensaje si hay más de 5 jugadores almacenados en la base de datos podríamos hacerlo del siguiente modo:

```
DECLARE
    NumJugadores NUMBER;
    E_MUCHOS_JUGADORES EXCEPTION;
BEGIN
    SELECT COUNT(*)
        INTO NumJugadores
        FROM JUGADOR;



    IF NumJugadores > 5 THEN
        RAISE E_MUCHOS_JUGADORES;
    END IF;
EXCEPTION
    WHEN E_MUCHOS_JUGADORES THEN
         DBMS_OUTPUT.PUT_LINE('Hay demasiados jugadores');
END;

Hay demasiados jugadores

Procedimiento PL/SQL terminado correctamente.
```

Finalmente, y dado que una excepción provoca un error que hace que un proceso termine anormalmente (haciendo un rollback, con lo que los cambios se deshacen), Oracle ofrece la posibilidad de producir una excepción de usuario mediante el procedimiento `RAISE_APPLICATION_ERROR` en la que se indica un código de error (`SQLCODE`) y un mensaje (`SQLERRM`)explícitos:

```
BEGIN
    RAISE_APPLICATION_ERROR(-20000,'Excepción creada por el
usuario');

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Código: '||SQLCODE);
        DBMS_OUTPUT.PUT_LINE('Mensaje: '||SUBSTR(SQLERRM,11,100));
END; / Código: -20000 Mensaje:  Excepción creada por el usuario

Procedimiento PL/SQL terminado correctamente.
```

1.7 Declarar y usar subprogramas: procedimientos y funciones

La sección de declaraciones de un bloque PL/SQL puede declarar una subtarea común como un *subprograma* (o *subrutina*) con nombre. Las siguientes sentencias en el cuerpo del programa principal pueden luego llamar al subprograma.

PL/SQL soporta dos tipos de procedimientos de subprogramas: procedimientos y funciones. Un *procedimiento* es un subprograma que realiza una operación. Una *función* es un subprograma que procesa un valor y lo devuelve al programa que llamó a la función.

1.7.1 Declarar y usar un procedimiento

La sintaxis general para declarar un procedimiento es la siguiente:

```
PROCEDURE procedimiento
    [(parámetro [IN|OUT|IN OUT] tipo_de_datos [{DEFAULT|:=}expresion] [,...])]
    IS
        declaraciones ...
    BEGIN
        sentencias ...
    END [procedimiento];
```

Cuando se declara un subprograma, como un procedimiento, se pueden pasar valores dentro y fuera del subprograma usando parámetros. Para cada parámetro debe especificar un tipo de datos de una forma no restringida. Además, debería indicar el modo de cada parámetro como IN, OUT o IN OUT:

- Un parámetro IN pasa un valor dentro del subprograma, pero un subprograma no puede cambiar el valor de la variable externa que corresponde a un parámetro IN.
- Un parámetro OUT no puede pasar un valor dentro de un subprograma, pero un subprograma puede manipular un parámetro OUT para cambiar el valor de la variable correspondiente en el entorno de la llamada externa.
- Un parámetro IN OUT combina las capacidades de los parámetros IN y OUT.

Veamos un ejemplo.

```
DECLARE
    PROCEDURE printLine(width IN INTEGER, chr IN CHAR DEFAULT '-')
    IS
    BEGIN
        FOR i IN 1 .. width LOOP
            DBMS_OUTPUT.PUT(chr);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('');
    END printLine;
BEGIN
    printLine(40, '*');                 - imprime una línea con 40 asteriscos
    printLine(width=>>20, chr=>>'=');   - imprime una línea con 20 signos igual
    printLine(10);                      - imprime una línea con 10 guiones
END;
```

El cuerpo del ejemplo llama al procedimiento printLine tres veces.

- La primera llamada proporciona valores para ambos parámetros del procedimiento usando la *notación posicional*: Cada valor de parámetro se corresponde con el parámetro del procedimiento declarado en la misma posición.
- La segunda llamada proporciona valores usando la *notación por nombre*: el nombre de un parámetro y el operador asociación (`=>>`) preceden al valor de un parámetro.
- La tercera llamada muestra que el usuario puede proporcionar valores para todos los parámetros del procedimiento sin valores predeterminados, pero puede omitir valores para los parámetros con valores predeterminados.

1.7.2 Declarar y usar una función

La sintaxis general para declarar una función es la siguiente:

```
FUNCTION función
    [(parámetro [IN|OUT|IN OUT] tipo_de_datos [{DEFAULT|:=} expresion] [,...])]
        RETURN tipo_de_datos
    IS
        declaraciones ...
    BEGIN
        sentencias ...
    END [funcion];
```

Como puede observarse, una función difiere de un procedimiento en que esta devuelve un valor (denominado *valor de retorno*) al entorno que hace la llamada. Además, el cuerpo de una función debe incluir una o más sentencias RETURN para devolver un valor al entorno que hace la llamada.

Ejemplo:

```
DECLARE
    tempPresup NUMBER;
    FUNCTION sumaPresup (codigo IN INTEGER)
        RETURN NUMBER
    IS
        sumaPresup NUMBER;
    BEGIN
        SELECT SUM(presupuesto) INTO sumaPresup
            FROM EQUIPO
            WHERE cp = codigo;
        RETURN sumaPresup;
    END sumaPresup;
BEGIN
    tempPresup := sumaPresup(15000);
    DBMS_OUTPUT.PUT_LINE('Presupuesto total Coruña'||tempPresup);
    DBMS_OUTPUT.PUT_LINE('Presupuesto total Madrid'||sumaPresup(28000));
END;
```

1.8 Procedimientos y funciones almacenados

Todo el código PL/SQL visto en este capítulo se ejecuta una sola vez, ya que no se "guarda" en ningún sitio. A continuación veremos cómo se pueden crear procedimientos y funciones (de forma similar a otros lenguajes de programación procedimentales) y almacenarlos en la base de datos, de forma que se hará una llamada al procedimiento o función para ejecutarlo.

Un **procedimiento** es un bloque de instrucciones PL/SQL que se almacena en el diccionario de datos y al que pueden llamar las aplicaciones. Los procedimientos permiten almacenar dentro de la BD la lógica de las aplicaciones que se emplea con más frecuencia. No devuelven ningún valor al programa que los llama. Los procedimientos almacenados pueden ayudar a forzar la seguridad de los datos, lo que se consigue no concediendo a los usuarios acceso directo a las tablas de una aplicación, sino sólo la capacidad de ejecutar un procedimiento que acceda a las tablas. Cuando se ejecuta el procedimiento, lo hará con los privilegios de su propietario. Los usuarios no podrán acceder a las tablas si no es por medio del procedimiento.

Las **funciones**, al igual que los procedimientos, son bloques de código que se almacenan en la BD. A diferencia de éstos, las funciones pueden devolver valores al programa que las llama. Es posible crear funciones e invocarlas desde las instrucciones SQL como si fuesen las propias funciones de Oracle, siempre que no modifique ninguna fila de la BD.

1.8.1 Crear y usar procedimientos almacenados

Para crear un procedimiento almacenado se usa el comando `CREATE PROCEDURE`. Su especificación es básicamente la misma que cuando se declara un procedimiento en la sección de declaraciones de un bloque PL/SQL.

Veamos un ejemplo:

```
CREATE OR REPLACE PROCEDURE printLine(width IN INTEGER, chr IN
CHAR DEFAULT '-')
    IS
    BEGIN
        FOR i IN 1 .. width LOOP
            DBMS_OUTPUT.PUT(chr);
        END LOOP;
        DBMS_OUTPUT.PUT_LINE('');
    END printLine;
```

Ahora puede usarse el procedimiento `printLine` en cualquier otro programa PL/SQL simplemente llamándolo:

```
BEGIN
    printLine(40, '*');                 - imprime una línea con 40 asteriscos
END;
/
BEGIN
    printLine(width=>>20, chr=>>'=');   - imprime una línea con 20 signos igual
END;
/
BEGIN
    printLine(10);                      - imprime una línea con 10 guiones
END;
/
```

1.8.2 Crear y usar funciones almacenadas

Para su creación se utiliza el comando `CREATE FUNCTION`. Se especifica como se haría en la sección de declaraciones de un bloque PL/SQL. Debe declararse el tipo que devuelve la función, y usar una o más sentencias `RETURN` en el cuerpo de la función para devolver el valor de retorno de la función.

Veamos un ejemplo:

```
CREATE OR REPLACE FUNCTION sumaPresup (codigo IN INTEGER)
        RETURN NUMBER
    IS
        sumaPresup NUMBER;
    BEGIN
        SELECT SUM(presupuesto) INTO sumaPresup
            FROM EQUIPO
            WHERE cp = codigo;
        RETURN sumaPresup;
    END sumaPresup;
```

Ahora, para recuperar las ciudades cuyo presupuesto total sea superior a 3000000 podría hacerse:

```
    SELECT *
        FROM CIUDAD
        WHERE sumaPresup(cp) > 3000000;
```