Sentencia SELECT

SELECT [DISTINCT] listaColumnas

FROM listaTablas

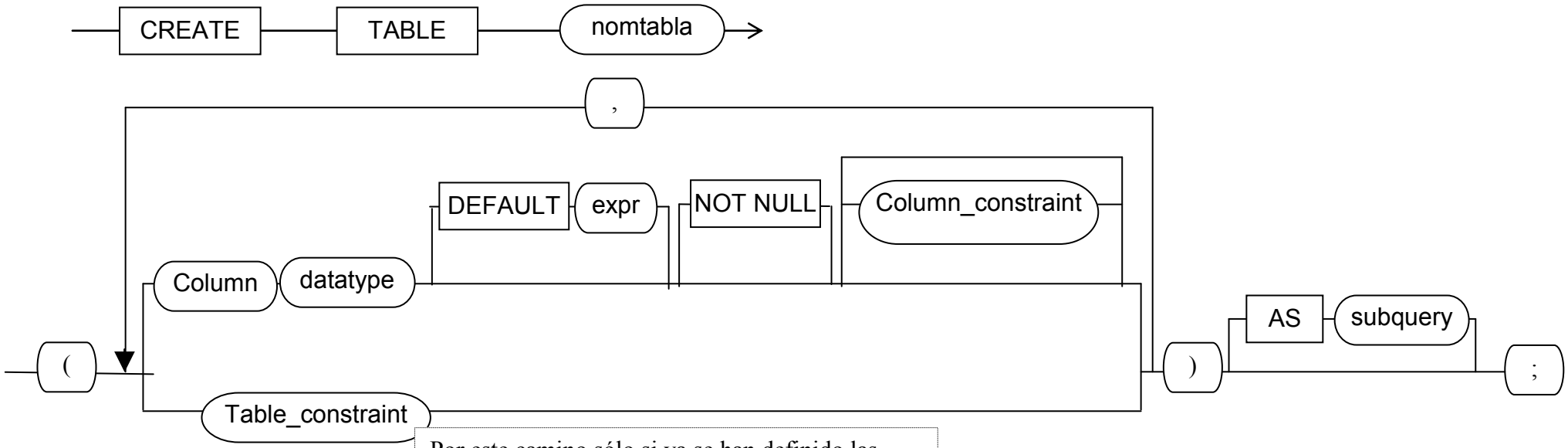
[**WHERE** condición para filas]

[**GROUP BY** listaColumnas por las que se quiere agrupar

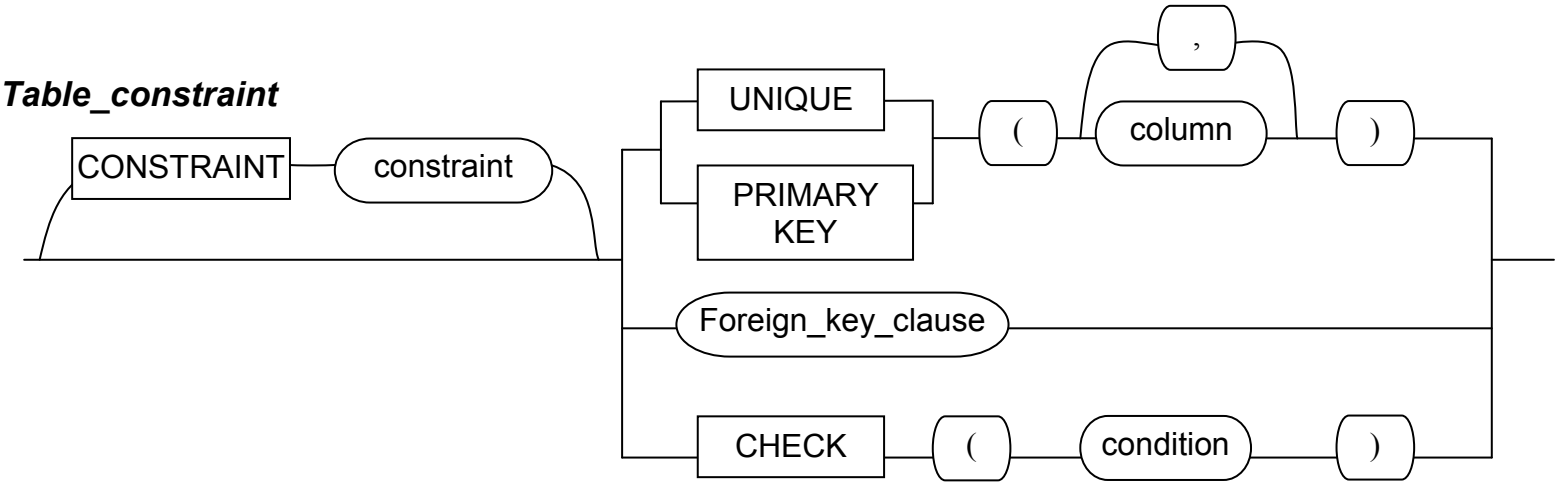
[**HAVING** condición para los grupos]]

[ORDER BY listaColumnas [ASC | DESC]]

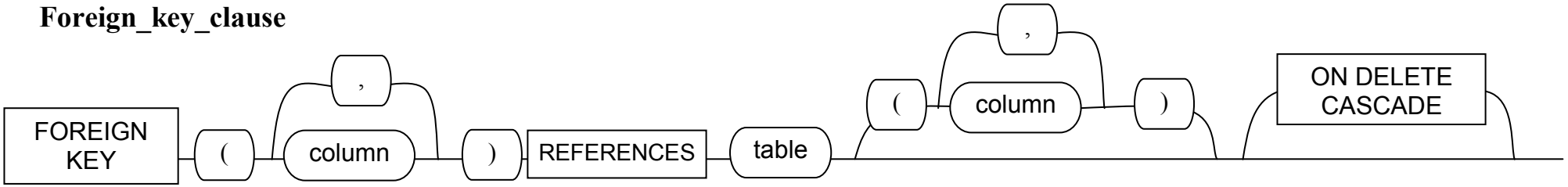
CREAR una TABLA



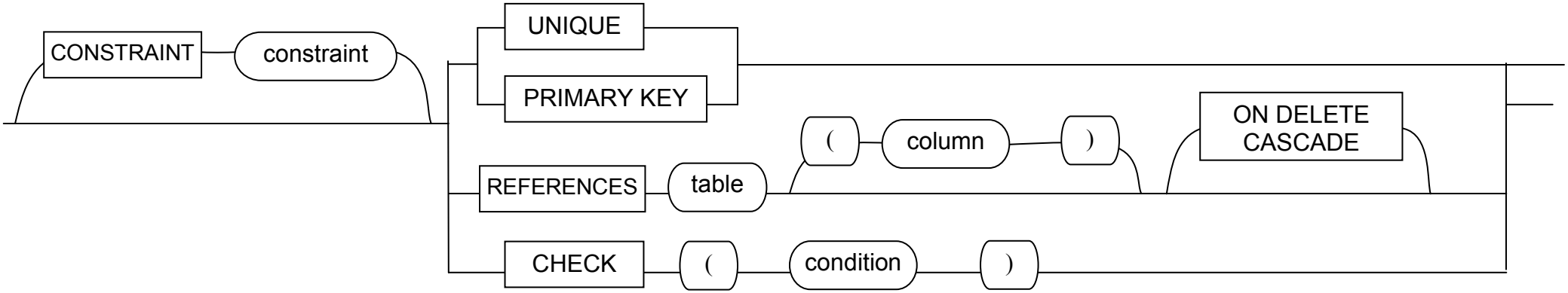
Table_constraint



Foreign_key_clause

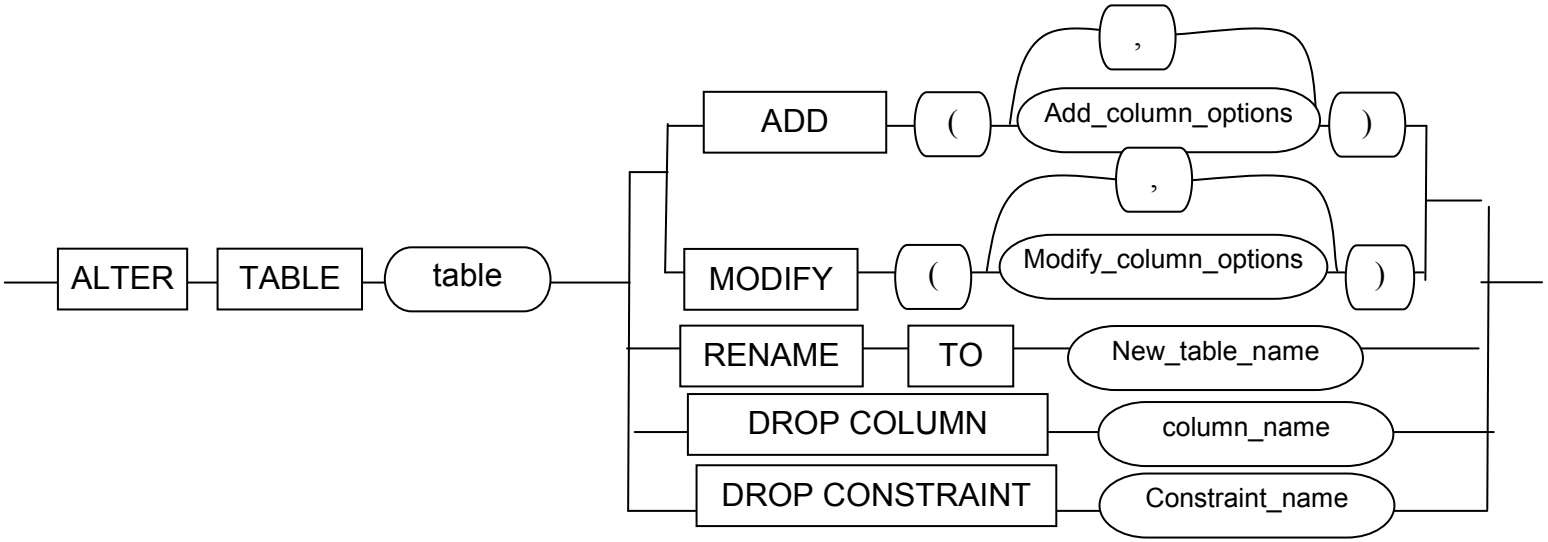


Column_constraint

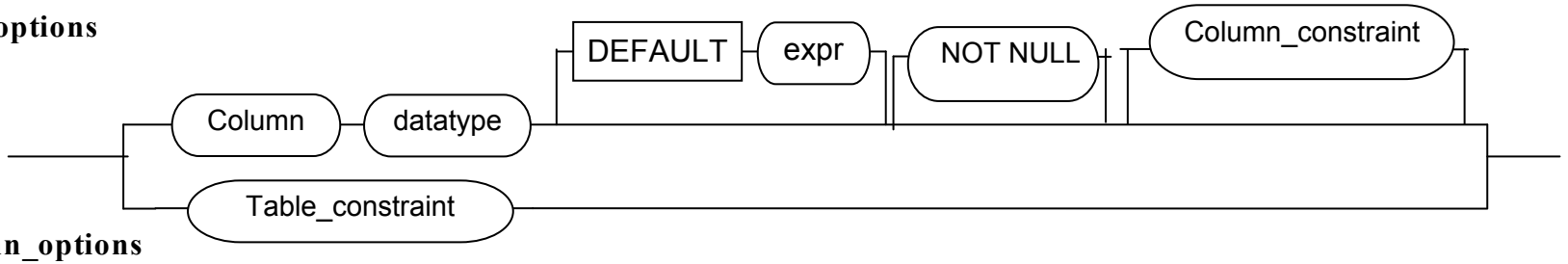


BORRAR UNA TABLA

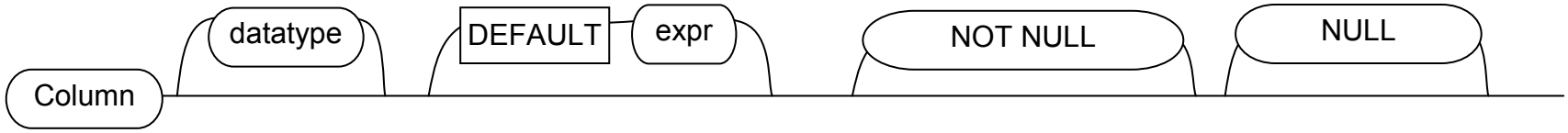
MODIFICAR la ESTRUCTURA y RESTRICCIONES de las TABLAS



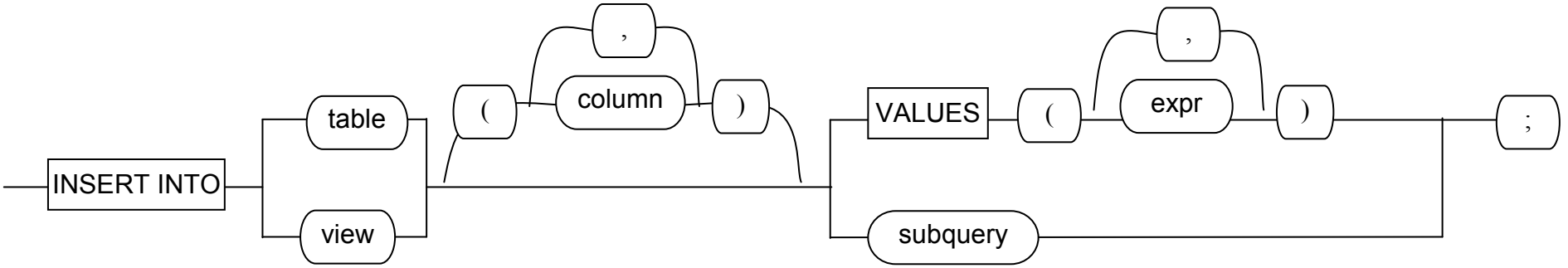
Add_column_options



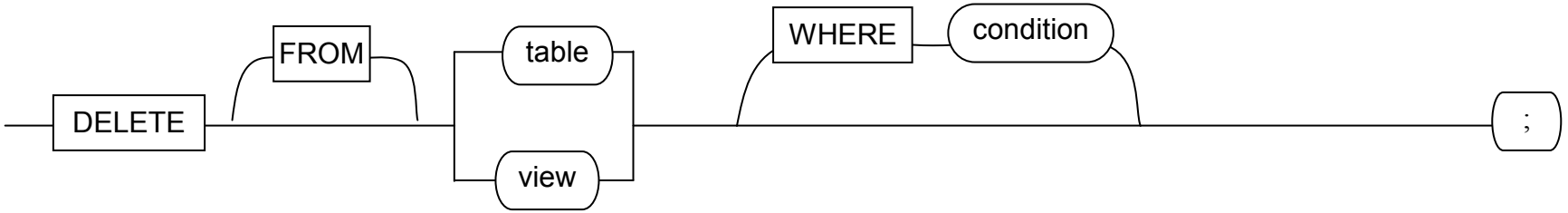
Modify_column_options



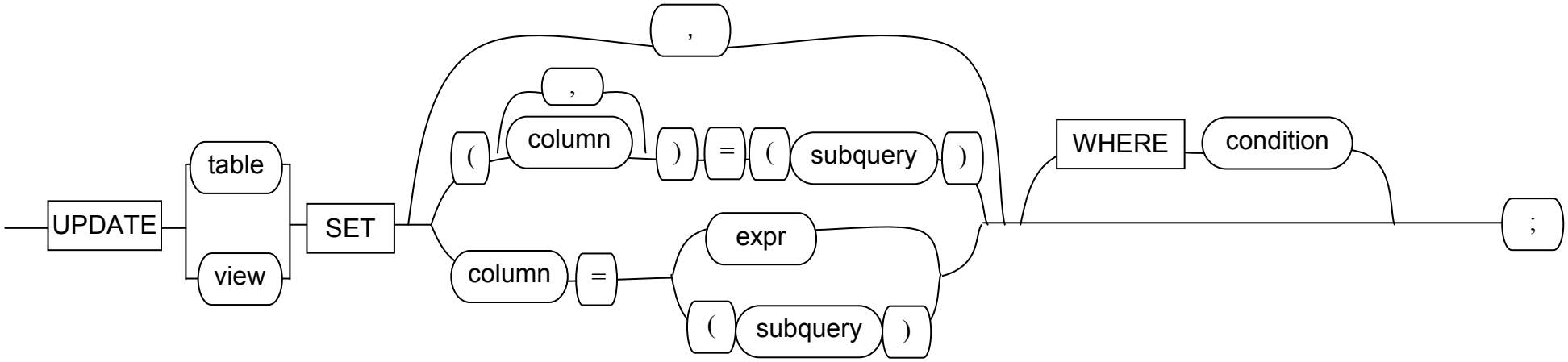
INSERTAR FILAS en una TABLA



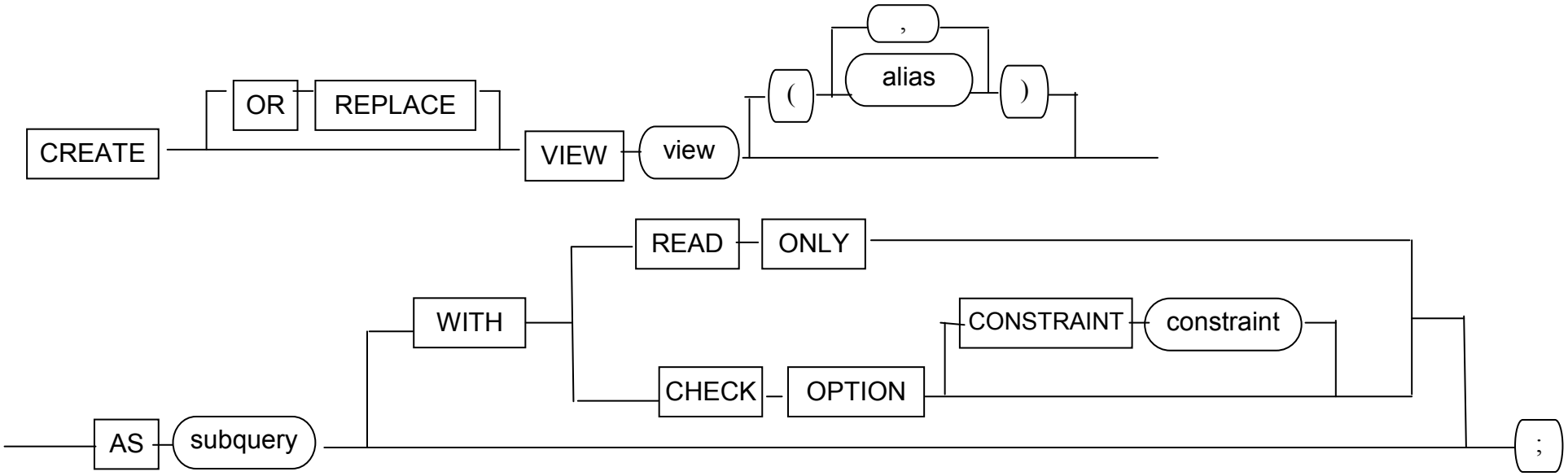
BORRAR FILAS de una TABLA



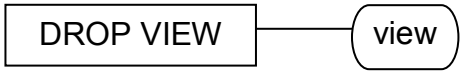
MODIFICAR los VALORES ALMACENADOS en una TABLA



CREATE VIEW



Para borrar vistas



MENSAJES.
DEFINICIÓN DE CONSTANTES, VARIABLES Y CURSORES.

Para mostrar mensajes podemos utilizar:

MENSAJES

raise_application_error (número_error, mensaje)
número_error –20000…-20999

raise_application_error(-20101,'Alcanzado total de votantes')

Muestra el mensaje y genera un error, interrumpiendo la ejecución.

dbms_output.put_line(mensaje)

Deposita el mensaje en un buffer y prosigue con la ejecución

Para que los mensajes se muestren es necesario ejecutar en la sesión de trabajo

SET SERVEROUTPUT ON FORMAT

A la hora de componer los mensajes se utiliza el operador concatenación ||

Lo podemos usar para construir los mensajes tanto con raise_application_error como con dbms_ouput.put_line

BLOQUES SQL

Un bloque SQL tiene tres partes: una declarativa, una ejecutable y una última para el manejo de instrucciones (warnings y condiciones de error). De estas partes, sólo la parte ejecutable es obligatoria.

```
[ DECLARE
        -- declaraciones]
BEGIN
        --  sentencias
[EXCEPTION
        --  manejo de excepciones]
END;
```

DECLARACIÓN de VARIABLES y CONSTANTES

En la parte declarativa se pueden declarar variables y constantes. Por ejemplo:

```
DECLARE
total number(6,2);
nombre varchar(30):= 'JUANA';
maximo CONSTANT number(4):=9999;
```

En la parte declarativa, las variables se pueden inicializar a un valor específico o no hacerlo, mientras que las constantes han de ser inicializadas en la parte declarativa.

DECLARACIÓN de VARIABLES y CONSTANTES

***Si necesitamos que el tipo de datos coincida con el definido para una columna de una tabla podemos utilizar* %TYPE**

El atributo %TYPE proporciona el tipo de dato de una variable o columna de la base de datos.

```
DECLARE
    valor number(4,2);
    auxvalor valor%TYPE;
    auxcategoria habitacion.categoria%TYPE;
```

La variable auxvalor es del mismo tipo de datos que valor y
la variable auxcategoria tiene el mismo tipo de datos que la columna categoría de la tabla habitación.

Utilizar el atributo %TYPE hace que no sea necesario conocer el tipo de dato exacto de una variable o columna, y por otro lado, tiene la ventaja de que si la definición del tipo de datos de una columna cambia, el tipo de dato de la columna también cambia automáticamente.

***Si necesitamos definir una estructura similar a una fila de una tabla podemos utilizar* %ROWTYPE**

El atributo %ROWTYPE obtiene un tipo de registro que representa una fila de una tabla. Los campos del registro y las correspondientes columnas de la tabla tienen el mismo nombre y el mismo tipo de datos

```
DECLARE
    pvp  pvptemporada%ROWTYPE;
```

ASIGNAR VALOR A UNA VARIABLE

Por defecto las variables se inicializan a NULL.
Se les puede asignar valor de dos formas: a través de una **expresión**, o a través de una **sentencia SELECT**.

```
aux_valor1:=25;
aux_valor2:=total-10;
aux_nombre:='ESTEBAN'
```

```
SELECT categoria  INTO auxcategoria  FROM habitacion WHERE ….
SELECT cod, descripción  INTO auxcod, auxdesc  FROM ACTIVIDADES WHERE …
```

Oracle utiliza áreas de trabajo para ejecutar sentencias SQL y almacenar la información procesada.
El uso de cursores permite dar nombre a un área de trabajo y acceder a la información almacenada en ella.

Cuando se declara un cursor, se le da un nombre, y se le asigna a una consulta específica.

CURSOR nombre_cursor IS sentencia_select;

Ejemplo:

```
DECLARE
CURSOR c1 IS SELECT num, pSA FROM habitación h, pvptemporada p  WHERE h.categoria=p.categoria;
```

Ahora es como si el resultado de ejecutar la sentencia SELECT fuera una tabla llamada c1 que podemos ir recorriendo fila a fila.

Trabajando con OPEN, FETCH y CLOSE *(aunque esta forma de trabajar con cursores es válida, **recomendamos** trabajar con cursores de un modo más sencillo, trabajando con **CURSORES en BUCLE FOR**, que se explica en el siguiente apartado)*

Para trabajar con cursores se pueden utilizar los comandos **OPEN, FETCH y CLOSE**.

OPEN
Abre el cursor, es decir, ejecuta la consulta e identifica el resultado (las filas resultantes de la consulta).
Al ejecutar OPEN, las filas no se devuelven.

```
DECLARE
        auxnum      habitación.num%type;
        auxsuperf   habitación.superf%type;
        auxsupMin   categoría.supMin%type;
        CURSOR c1 IS SELECT num, superf, supMin FROM habitación , categoría   WHERE categoria=nombre;
BEGIN
        OPEN c1;
        …
        FETCH c1 INTO auxnum, auxsuperf, auxsupMin;
        …
        CLOSE c1;
        …
END;
```

FETCH
Permite devolver las filas del resultado.
Cada vez que se ejecuta FETCH el cursor avanza a la siguiente fila del resultado.

CLOSE
Cierra el cursor. Una vez cerrado podría volver a abrirse.

Para cada columna que se devuelve en la consulta asociada al cursor, tendremos que tener una variable después del INTO, con un tipo de datos compatible.

Cada cursor tiene asociados cuatro atributos**: %FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT**

Tras abrir un cursor con OPEN y antes de que se haya hecho el primer FETCH

- el atributo %FOUND contiene NULL
- el valor de %ROWCOUNT es 0

A partir de ese momento cada vez que hacemos FETCH,

- %FOUND devolverá TRUE si el último FETCH devolvió una fila, o FALSE en caso contrario
- %ROWCOUNT contendrá el número de filas a las que ya se ha realizado FETCH.

El atributo opuesto a %FOUND es %NOTFOUND.

El atributo %ISOPEN devuelve

- true si el cursor está abierto
- false si no está abierto

Ejemplo:

Si c1 es un cursor que hemos definido, se podrían utilizar expresiones como

```
IF c1%FOUND THEN …
IF c1%ROWCOUNT >4 THEN …
```

Al trabajar con cursores, se puede simplificar el código utilizando un **bucle FOR en lugar de** OPEN, FETCH y CLOSE. El bucle FOR:

- **abre implícitamente el cursor**
- **realiza FETCH repetidamente**
- **y cierra el cursor cuando todas las filas han sido procesadas.**

En el bucle además de abrir el cursor se declara una variable (*en el ejemplo que sigue* regc1).

- Esta variable sólo puede ser utilizada dentro del bucle.
- Es una variable de tipo registro, cuyos campos tienen el mismo nombre y tipo de datos que las columnas de la sentencia SELECT que figura en la definición del cursor.
- Si alguna de las columnas fuese calculada sería necesario que tuviese un alias en la sentencia SELECT

Ejemplo:

```
DECLARE
    •
    CURSOR c1 IS SELECT num, superf, supMin FROM habitación , categoría   WHERE categoria=nombre;
    •
    •
BEGIN
    •
    •
    FOR regc1 IN c1 LOOP
        IF regc1.sup < regc1.supMin THEN
            Dbms_output.put_line(' Habitación '|| regc1.num ||
             ' dimensión INCORRECTA ***');
            INSERT INTO MALCATEGORIA
                VALUES(regc1.num, regc1.supMin-regc1.superf);
        ELSE
            Dbms_output.put_line(' Habitación '|| regc1.num ||
             ' dimensión correcta');
        END IF;
    END LOOP;
    •
    •
    •
END;
```

Suponemos creada MALCATEGORIA(num: number(3), dif: number(4,2))

En el bloque DECLARE junto con la definición de variables y constantes, se hace la declaración de los cursores

En el bloque ejecutable (BEGIN-END) se puede abrir el cursor y trabajar con él.

En la primera iteración:
- se abre (OPEN) el cursor,
- se accede a la primera fila (FETCH) y
- se deposita en regc1 (queda declarada en el FOR)

Cada nueva iteración significa el acceso a la siguiente fila (FETCH) y depositarla en regc1.

Al terminar de recorrer todas las filas, finalizan las iteraciones y se cierra automáticamente el cursor (CLOSE)

Para trabajar dentro del bloque BEGIN-END se pueden utilizar estructuras de control similares a las de otros lenguajes de programación.

ESTRUCTURAS DE CONTROL CONDICIONAL

```
IF … THEN…
END IF;
```

```
IF … THEN…
ELSE …
END IF;
```

```
IF … THEN…
ELSIF … THEN …
ELSIF … THEN …
[ELSE …]
END IF;
```

ESTRUCTURAS DE CONTROL ITERATIVO

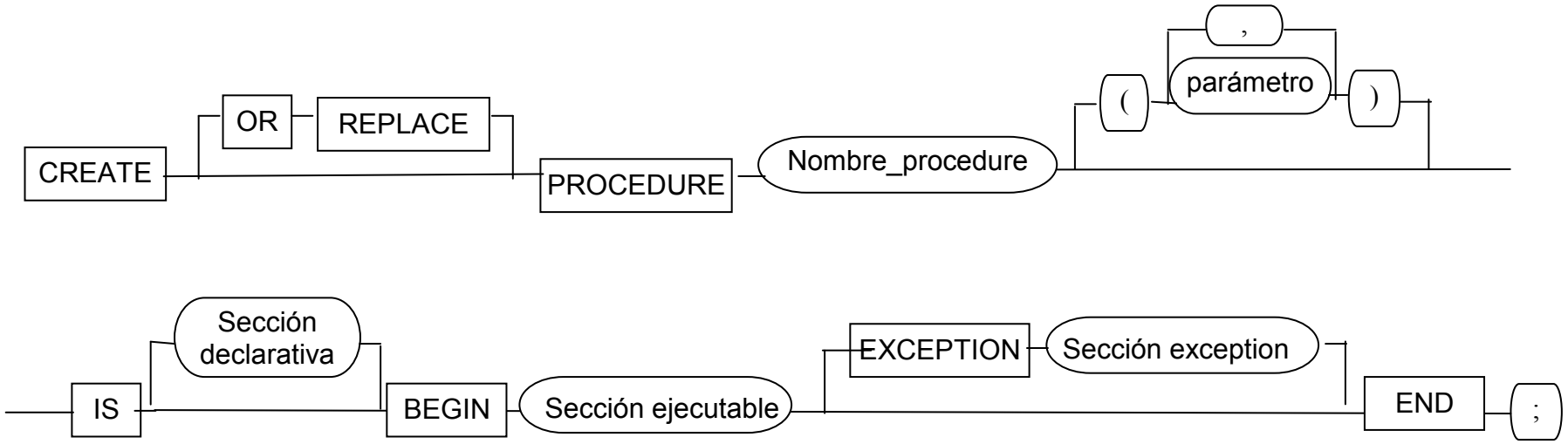
```
FOR i IN min .. max LOOP
  …
  …
END LOOP;
```

```
WHILE condición LOOP
  …
END LOOP;
```

```
LOOP
 …
…
…
EXIT WHEN …;
END LOOP;
```

```
LOOP
 …
   IF condición THEN
         …
         EXIT;
   END IF;
…
END LOOP;
```

CREATE PROCEDURE



Al definir los parámetros se debe poner*: nombre_parámetro tipo_de _parámetro tipo_de_datos(Sin longitud)*
Como tipo de parámetro puede ser:

- IN(por defecto) pudiendo pasarle valores en la llamada al procedimiento. Este valor no se puede modificar. Se le puede asignar un valor por defecto.

  CREATE PROCEDURE EJEMPLO(entrada in Lumber default 10) IS …

- OUT para devolver valores.
- IN OUT que permite pasarle valores en la llamada al procedimiento y luego devolver valores.

Para borrar un procedimiento **DROP PROCEDURE nombre_procedure**

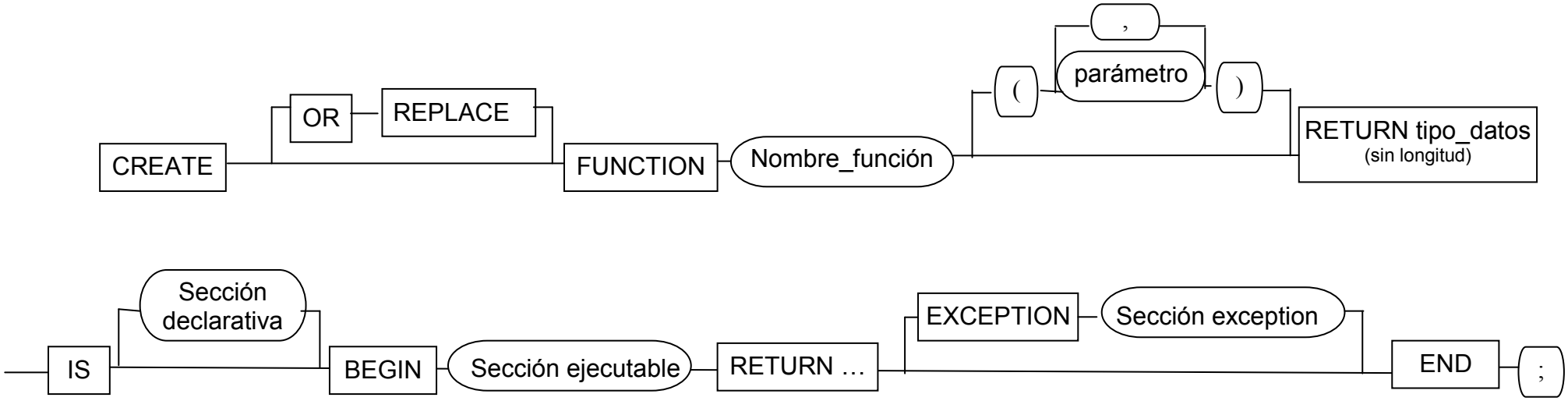
Ejemplo:

```
create or replace procedure verminsuperficie  is

    CURSOR c1 IS
    SELECT numero, superficie, supMin FROM habitación , categoría   WHERE categoria=nombre;
 BEGIN

    FOR regc1 IN c1 LOOP
        IF regc1.sup < regc1.supMin THEN
            Dbms_output.put_line(' Habitación '|| regc1.num || ' dimensión INCORRECTA ***');
            INSERT INTO MALCATEGORIA
                VALUES(regc1.num, regc1.supMin-regc1.superf);
        ELSE
            Dbms_output.put_line(' Habitación '|| regc1.num || ' dimensión correcta');
        END IF;
    END LOOP;

END;
```

Se puede ejecutar directamente **exec** verminsuperficie;
o bien desde dentro de otro procedimiento

Tal y como se ha hecho en clase, al crear un procedure, una función o un trigger(a continuación) con SQL-Developer, os situáis sobre él y seleccionáis la opción COMPILE. **Si se os indica que se ha creado con errores de compilación debéis corregirlos**.

CREATE FUNCTION



Al definir los parámetros se debe poner*: nombre_parámetro tipo_de _parámetro tipo_de_datos(Sin longitud)*
Como tipo de parámetro puede ser:

- IN(por defecto) pudiendo pasarle valores en la llamada a la función. Este valor no se puede modificar. Se le puede asignar un valor por defecto.

  CREATE FUNCTION EJEMPLO(entrada in number default 10) RETURN varchar IS …

- Los tipos OUT e IN OUT no se suelen utilizar en las funciones.

Para borrar una función **DROP FUNCTION nombre_función**

Para ejecutar directamente una función se puede hacer

Exec escribir(nom_función(param1)); (aunque el parámetro de entrada de escribir es varchar, hace la transformación de number y date a varchar)

SELECT nom_función(param1) FROM dual;

También se puede llamar desde:

- Un procedimiento
- Una sentencia SELECT, tanto como columna en la SELECT como en las condiciones de las cláusulas WHERE y HAVING como en el GROUP BY.
- Una sentencia INSERT dentro de los VALUES.
- Una sentencia UPDATE en la cláusula SET.

Ejemplo:

```
create or replace function edad (eldni char) return number
is
aux number(2);
begin
select (to_char(sysdate,'yyyy')-to_char(fechanac,'yyyy')) into aux
from empleados where dni=eldni;
return(aux);
end;
```

Se puede ejecutar directamente

```
Select edad('11111111A')  from dual;
```

O bien

```
select count(*), edad(dni) from empleados group by edad(dni);
```

O bien dentro de un procedimiento

```
create or replace procedure listadoporactividad(laactividad char, laedad number) is
cursor emp  is select e.dni dni,nombre, direccion
                from empleados e, animacion an, actividad ac
               where e.dni= a.dni and ac.empleado=an.dni;
begin
escribir('EMPLEADOS MENORES de '||laedad||'  DE LA ACTIVIDAD '||laactividad||' : ');
for e in emp loop
if edad(e.dni)<laedad then
escribir('Nombre: '||e.nombre||'   dirección   '||e.direccion);
end if;
end loop;
end;
```

```
exec listadoporactividad('AC10',50);
```

En la definición de los cursores debemos dar nombre a las columnas en las que se utilicen alias de las tablas o las columnas que sean funciones agregadas.
(En el ejemplo no necesariamente hay que llamarla "dni "… podría ser empleado o cualquier otro nombre representativo)

MANEJO DE EXCEPCIONES

En la mayoría de los lenguajes de programación se emplean EXCEPCIONES para tratar errores en tiempo de ejecución. En Oracle también podemos definir excepciones. Estas definiciones hacen que, cuando se produce un error, salte una excepción y pase el control a la sección excepción correspondiente. Las acciones a seguir se incluyen en la sección EXCEPTION definida al final del bloque BEGIN-END.

```
BEGIN

  …

EXCEPTION
   WHEN <nombre_excepción> THEN
      <bloque de sentencias>;
  WHEN <nombre_excepción> THEN
      <bloque de sentencias>;

   ...
   [WHEN OTHERS THEN <bloque de sentencias>;]
END;
```

Existen dos tipos de excepciones:
las excepciones predefinidas y las excepciones definidas por el usuario.

EXCEPCIONES PREDEFINIDAS (Al estar ya definidas en Oracle no se declaran en la sección DECLARE)

Son aquellas que se disparan automáticamente al producirse determinados errores. Las más frecuentes son:

- **too_many_rows**: Se produce cuando SELECT … INTO devuelve más de una fila.
- **no_data_found**: se produce cuando un SELECT… INTO no devuelve ninguna fila.
- **value_error**: se produce cuando hay un error aritmético o de conversión.
- **zero_divide**: se puede cuando hay una división entre 0.
- **dupval_on_index**: se crea cuando se intenta almacenar un valor que crearía duplicados en la clave primaria o en una columna con restricción UNIQUE.
- **invalid_number**: se produce cuando se intenta convertir una cadena a un valor numérico.

EXCEPCIONES DEFINIDAS POR EL USUARIO

Son las que define el usuario. Para poder trabajar con excepciones definidas por el usuario, es necesario que se realicen 3 pasos:

1. Definición del nombre de la excepción en la sección DECLARE. La sintaxis es

   Nombre_de_excepción EXCEPTION

2. Lanzar la excepción. Se hace mediante la orden RAISE ;
3. Definir las acciones a seguir si se lanza la excepción. Esto se hace en la sección EXCEPTION.

   WHEN nombre_de_excepción THEN …

Ejemplo :

Crea una función en la que dado un número de habitación devuelva su categoría.

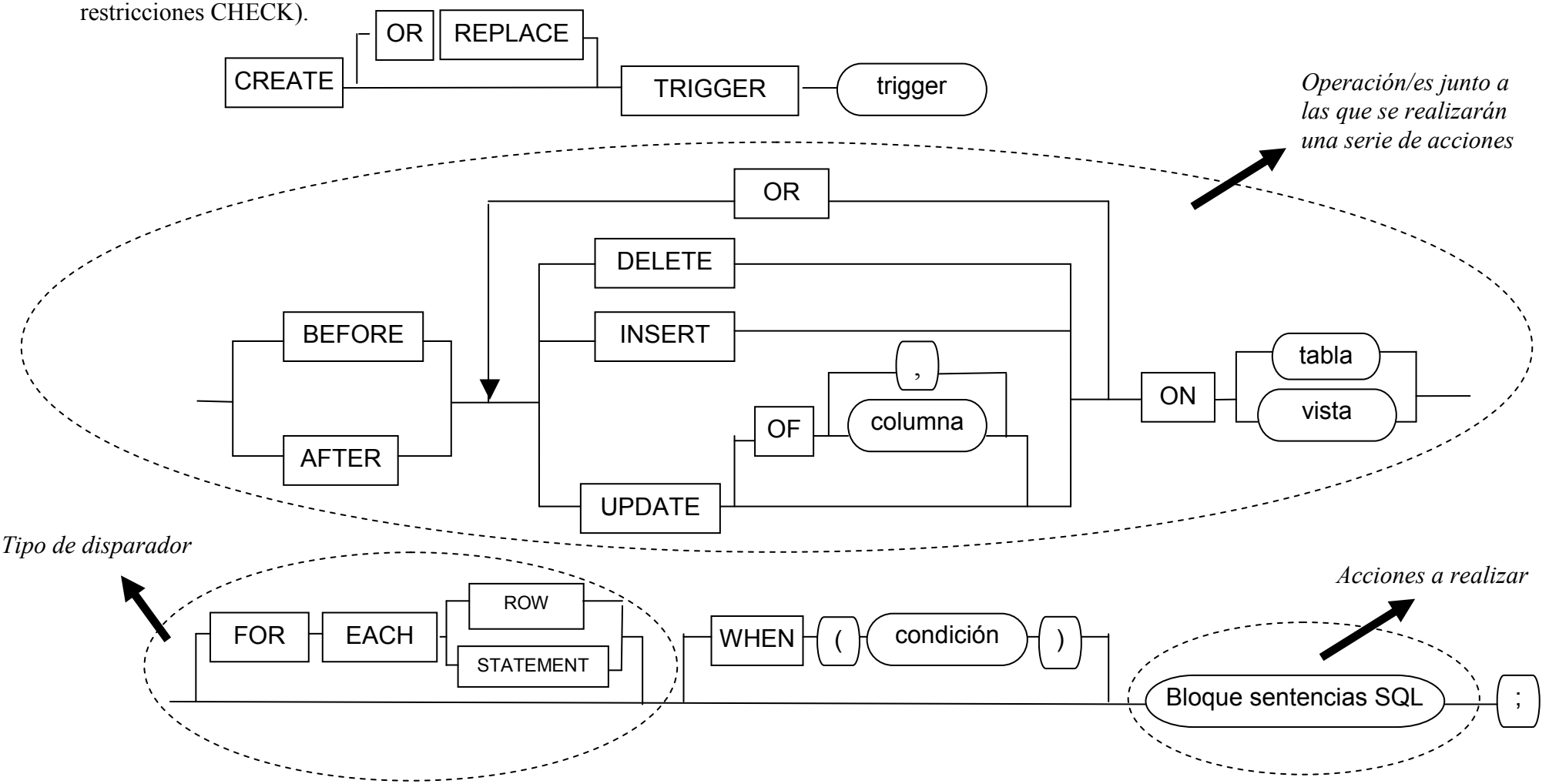
```
create or replace function tipo_habitacion (elnumero habitacion.numero%type) return habitacion.categoria%type
is
auxcategoria habitacion.categoria%type;
begin
select categoria into auxcategoria
from habitacion
where numero=elnumero;
return(auxcategoria);
exception
  when no_data_found then
     escribir('No existe ninguna habitación con ese número');
     return null;
end;
```

Ejemplo :Crea un procedimiento en el que dado un nif de un empleado y un valor X (número). Incluya a ese empleado como recepcionista siempre que no existan ya más de X recepcionistas. Si al darlo de alta como recepcionista vemos que no está dado de alta como empleado, se dará de alta tanto de empleado como de recepcionista y se mostrará un mensaje indicando que falta por completar sus datos.

```
create or replace procedure pon_recepcion(elnif empleado.nif%type, maximo number) is
      auxnombre empleado.nombre%type;
      auxtelefono empleado.telefono%type;
      total number(3);
      pasa_tope exception;
BEGIN
 select count(*) into total from emprecepcion;
 if total>maximo then raise pasa_tope; end if;
 select nombre, telefono into auxnombre, auxtelefono
 from empleado where nif=elnif;
 escribir('Nombre '||auxnombre||'  pasa a tareas de recepcionista');
 insert into emprecepcion values(elnif);
EXCEPTION
    WHEN no_data_found then
        insert into empleado(nif) values(elnif);
        insert into emprecepcion values(elnif);
        escribir('No existía. Se ha dado de alta como recepcionista, falta   completar  sus datos en empleado');
    WHEN pasa_tope then
      escribir('Se supera el tope asignado para recepcionistas');
END;
```

CREATE TRIGGER

Crear un trigger asociado a una operación de manipulación de datos sobre una tabla T, significa definir una serie de acciones que se desencadenarán automáticamente cuando se realice esa operación sobre la tabla T. Puede ser útil para completar los valores de columnas derivadas(cuyo valor se calcula partiendo del de otras columnas) o para establecer restricciones complejas (no se han podido establecer mediante restricciones CHECK).



Dentro del trigger se pueden usar predicados condicionales para ejecutar distinto código según el tipo de sentencia que ha activado el trigger.

- INSERTING devuelve verdad si el trigger se ha activado por una sentencia INSERT
- DELETING devuelve verdad si el trigger se ha activado por una sentencia DELETE
- UPDATING devuelve verdad si el trigger se ha activado por una sentencia UPDATE
- UPDATING('nom_colu') devuelve verdad si el trigger se ha activado por una sentencia UPDATE y nom_colu se ha actualizado.

Cuando realizamos operaciones **INSERT** o **UPDATE**, para hacer referencia a los nuevos valores utilizaremos **:new**.

Por ejemplo, si estamos insertando en votantes y queremos referirnos al nif de cada fila insertada, dentro de FOR EACH ROW nos referiremos al nif con :new.nif .

Al realizar operaciones **DELETE** o **UPDATE**, para hacer referencia a los valores que hemos eliminado utilizaremos **:old**. (:old.nif)

Cuando se utiliza la condición **WHEN** al crear un trigger, las variables new y old van sin los:

Se recuerda que tanto al crear una función, un procedimiento o un trigger hay que verificar que se ha creado sin errores de compilación y, en caso de que los haya, mirar los que son y corregirlos.

Ejemplo 1

Trigger para controlar que la generalización de EMPLEADO sea DISJUNTA.
Se muestra como ejemplo el trigger que controla que los empleados que insertemos en ***emprecepcion*** *no estén ya en las tablas* ***empanimacion***, ***emplimpieza***, ***emprestaurante*** *y* ***empservicios****. Para reflejar que esa generalización es disjunta habría que completar con triggers similares en las tablas* ***empanimacion***, ***emplimpieza***, ***emprestaurante*** *y* ***empservicios.***

```
create or replace trigger disj_emprecepcion
before insert or update of nif
on emprecepcion
for each row
declare cuantos number(1);
begin
cuantos:=0;
select count(*) into cuantos from empanimacion where nif=:new.nif;
if (cuantos=1)
  then raise_application_error(-20601,'El nif '||:new.nif || ' ya es empleado de animación');
end if;
select count(*) into cuantos from emplimpieza where nif=:new.nif;
if (cuantos=1)
  then raise_application_error(-20601,'El nif '||:new.nif || ' ya es empleado de limpieza');
end if;
select count(*) into cuantos from emprestaurante where nif=:new.nif;
if (cuantos=1)
  then raise_application_error(-20601,'El nif '||:new.nif || ' ya es empleado de restaurante');
end if;
select count(*) into cuantos from empservicios where nif=:new.nif;
if (cuantos=1)
then raise_application_error(-20601,'El nif '||:new.nif || ' ya es empleado de servicios');
end if;

end;
```

Ejemplo 2

Tenemos una tabla

COMPROBAR_ESTUDIOS(NIF char(9), estudios varchar(50))

C.P.: NIF

Clave ajena: NIF→ EMPLEADO

Queremos que cuando se inserten los datos de un empleado y tenga valor en la columna estudios, insertemos en la tabla COMPROBAR_ESTUDIOS el NIF y los estudios que indica el empleado y lo comuniquemos.

```
create or replace trigger control_estudios
after insert on empleado
for each row
when (new.estudios is not null)
begin
 insert into comprobar_estudios values(:new.nif, :new.estudios);
escribir('Se comprobarán los estudios de  '|| :new.nombre);
end;
```