Département Informatique
1ème année

Bases de Données
Le langage PL/SQL d'ORACLE

Rosine Cicchetti

Sommaire

Préambule 5

1 - Structure d'un bloc PL/SQL 6

2 - Partie déclarative d'un bloc PL/SQL 7

2.1 - Les variables scalaires 7
2.1.1 - Déclaration de variables 7
2.1.2 - Déclaration basée ou par référence 8

2.2 - Les constantes 9

2.3 - Les variables composées 9
2.3.1 - Déclaration de variables enregistrements 9
2.3.2 - Déclaration basée ou par référence de variables enregistrements 11
2.3.3 - Déclaration de variables tableaux 12

2.4 - Les exceptions 13

3 - Instructions PL/SQL 13

3.1 - Affectation 13
3.1.1 - Affectation classique 14
3.1.2 - Affectation par extraction de données de la base 15

3.2 - Instruction conditionnelle 16

3.3 - Itérations 18
3.3.1 - Boucle simple 18
3.3.2 - Boucle FOR 19
3.3.3 - Boucle WHILE 20

3.4 - Autres instructions 20

4 - Gestion des exceptions 21

4.1 - Les exceptions système nommées et anonymes 21

4.2 - Déclenchement des exceptions utilisateur nommées 23

4.3 - Déclenchement des exceptions utilisateur anonymes 23

4.4 - Traitement des exceptions 23

5 - Ordres SQL valides en PL/SQL 24

6 - Les curseurs 25

6.1 - Définition de curseur 26

6.2 - Gestion de curseurs 27

6.3 - Propriétés de curseurs 28

6.4 - Parcours automatique de curseur 30

6.5 - Paramétrer des curseurs 31

7 - Blocs anonymes, procédures et fonctions 32

7.1 - Spécification de blocs anonymes 32

7.2 - Création et utilisation de procédures 34
7.2.1 - Procédures locales 36
7.2.2 - Procédures stockées 36

7.3 - Création et utilisation de fonctions 38

7.3.1 - Fonctions locales 39
7.3.2 - Fonctions stockées 39
7.4 - Surcharge des procédures et fonctions 39
7.5 - Affichage de messages 40
8 - Les packages 40
9 - Compilation et suppression de blocs stockés 43
10 - SQL dynamique natif 43
10.1 - Mises à jour et ordres SELECT simples 43
10.1.1 - Utilisation de EXECUTE IMMEDIATE 45
10.1.2 - Curseurs implicites 47
10.1.3 - Appel dynamique de blocs PL/SQL 47
10.2 - Ordres SELECT rendant un ensemble de résultats 48
11 - Conseils pratiques 49
11.1 - Écriture de programmes PL/SQL 49
11.2 - Mise au point de programmes PL/SQL 50
12 - Quelques références 52

Préambule

Développer des applications bases de données ne peut pas se faire en utilisant seulement SQL mais nécessite les capacités procédurales et les structures de données complexes offertes par les langages de programmation. Destiné aux développeurs, PL/SQL, abréviation de « Procedural Language extensions to SQL », est un langage de programmation de haut niveau largement inspiré des langages PASCAL et ADA. Il est proposé par ORACLE et permet de développer des traitements sur une base de données. Ces traitements peuvent être :

- des programmes, sous forme de procédures, fonctions, packages ou blocs anonymes ;
- des déclencheurs (ou triggers) qui sont des « parties de code » exécutées automatiquement par ORACLE ou SQLFORMS[1] lorsqu'un événement est détecté par le système.

Le code écrit en PL/SQL peut intégrer des ordres SQL. Ceci n'est pas spécifique à PL/SQL. En fait les SGBD commercialisés permettent de faire des développements dans la plupart des langages de programmation, qui sont alors qualifiés de langages hôtes. Un pré-compilateur effectue la traduction des ordres SQL dans le langage en question, avant la compilation.
Ainsi, dans de tels langages aussi bien qu'en PL/SQL, il est possible de tirer profit, au sein d'un programme, des capacités de traitements offertes par le SGBD (via SQL) et de les compléter, lorsqu'elles sont insuffisantes, par des fonctionnalités développées par un programmeur.
En fait, dans l'écriture d'un programme PL/SQL, ***il faut exploiter au maximum les possibilités offertes par le SGBD***. Il est par exemple inutile de réécrire des fonctions de conversion de types ou d'extraction de sous-chaînes mais il faut utiliser les fonctions disponibles en SQL et PL/SQL[2].
Si un programme doit travailler sur un ensemble de tuples résultats d'une requête et que ces tuples doivent être triés, il est inutile de re-développer un algorithme de tri mais il faut intégrer la clause **`ORDER BY`** dans la requête d'extraction…

Lorsqu'un développeur utilise un langage hôte, il travaille dans deux univers différents comme l'illustre la figure 1. Plus précisément, il s'agit de faire cohabiter dans la description d'un traitement d'une part une manipulation ensembliste de tuples via une requête SQL et d'autre part une gestion individuelle d'enregistrements, seule possibilité offerte par les langages procéduraux.

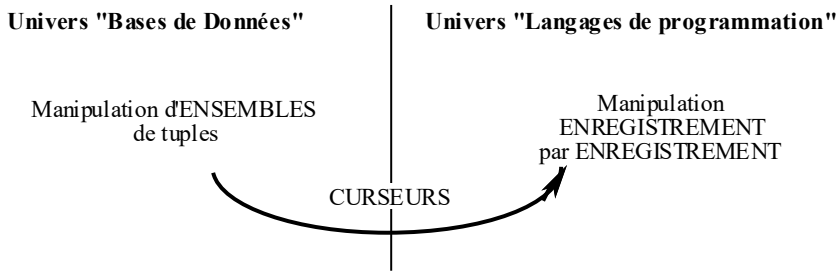

Figure 1 : Problématique du développement en langage hôte

La correspondance entre un tuple et un enregistrement semble naturelle. Néanmoins, pour assurer les échanges entre « l'univers BD » et « l'univers langage de programmation », il est nécessaire de disposer d'un outil permettant d'extraire des données de la base de manière ensembliste (en fait via une requête) et de travailler sur ces données de manière procédurale. Cet outil est un curseur

[1] Utilitaire de création de masques, ou plus généralement d'interfaces utilisateur, proposé par ORACLE.
[2] Toutes les fonctions de calcul horizontal disponibles en SQL sont également proposées par PL/SQL.

(Cf. paragraphe 6). D'autres problèmes se posent lors du développement en langage hôte, ne serait-ce que la compatibilité des types…

À propos de ce document :

Ce support de cours a été conçu pour des étudiants connaissant le modèle relationnel et le langage SQL*PLUS. Son objectif est une présentation de la programmation en PL/SQL. Sont mis en évidence les spécificités du développement « au-dessus d'une base de données » et le besoin de minimiser la maintenance des applications. Bien qu'écrits en PL/SQL, les déclencheurs ou triggers relevant de la programmation événementielle des applications bases de données ne sont pas abordés ici, ils font l'objet d'un autre support de cours.

Conventions :

Dans ce support, les mots clefs SQL et PL/SQL sont indiqués en police `courier`, en majuscules et en gras, les commandes ORACLE sont en `courier`, en minuscules et en gras. Les éléments optionnels des ordres sont indiqués entre crochets. Les éléments que doit spécifier l'utilisateur apparaissent entre < et >. L'alternative est représentée par : `|`. Tous les éléments issus directement ou non du schéma d'une base de données sont notés en majuscules, ceux qui sont spécifiés par le développeur en PL/SQL sont indiqués en minuscules. La base de données exemple est celle utilisée en TP.

1 - Structure d'un bloc PL/SQL

Un traitement PL/SQL est décrit sous forme d'un bloc qui peut être anonyme (le bloc est un script PL/SQL) ou nommé. Les blocs nommés sont des procédures, des fonctions ou des packages. La syntaxe générale de définition d'un bloc est la suivante :

```
[<specification_en_tete>]           -- Section Entête
[DECLARE | IS]
[<liste_déclarations>]              -- Section Déclarations

[BEGIN]
     <liste_instructions>           -- Section Instructions

[EXCEPTION
     <gestion_exceptions>]          -- Section Gestion des Exceptions

[END] ;
```

Remarques :

- Toute déclaration et toute instruction de n'importe quelle section se terminent par le caractère « ; ».
- La spécification d'un en-tête est inutile pour un bloc anonyme. Elle n'est utilisée que pour les procédures, les fonctions et les packages.
- La partie déclarative `<liste_déclarations>` n'est obligatoire que s'il y a des variables, constantes, curseurs, exceptions, types ou autres traitements (i.e. procédures ou fonctions) locaux au bloc.
- Le mot clef **`DECLARE`** ne doit être utilisé que pour introduire des déclarations dans les blocs anonymes. Le mot clef **`IS`** est obligatoire pour les procédures et les fonctions (qu'elles aient ou pas des déclarations locales).
- Dans un bloc anonyme, **`BEGIN`** et **`END`** ne sont obligatoires que s'il y a une partie déclarative. Dans un bloc nommé, **`BEGIN`** et **`END`** sont obligatoires.

- La partie <liste_instructions> est obligatoire et peut éventuellement être réduite à l'instruction « **NULL ;** » qui ne fait rien.
- La partie <gestion_exceptions> est optionnelle. Elle est utilisée pour traiter les anomalies détectées lors de l'exécution du bloc.
- Comme en SQL, les commentaires sont encadrés par /* et */ ou introduits par -- pour une seule ligne de commentaire.

Aux paragraphes 7 et 8, les spécificités des différents types de blocs sont décrites. D'ici là, sont détaillées les parties communes concernant tout bloc PL/SQL, i.e. les déclarations, la définition de curseurs, le corps du bloc constitué d'instructions, la gestion des exceptions.

2 - Partie déclarative d'un bloc PL/SQL

Les déclarations concernent les variables simples ou composées, les constantes, les types spécifiés par l'utilisateur, les curseurs, les exceptions, les procédures ou les fonctions locales. Tous ces éléments sont dotés d'un nom respectant les règles des identificateurs ORACLE : un identificateur est une chaîne de caractères dont la longueur maximale est 30, commençant par une lettre et pouvant inclure lettres, chiffres ainsi que des blancs soulignés, dollars et dièses.
Dans ce paragraphe, sont détaillées les déclarations de variables, types, constantes et exceptions. Les définitions de curseurs et de blocs locaux sont traitées respectivement aux paragraphes 6 et 7.

2.1 - Les variables scalaires

Les variables manipulées en PL/SQL peuvent être scalaires (simples). Elles doivent être nommées en respectant les règles indiquées ci-dessus. Comme pour les noms d'attributs en SQL, les variables doivent avoir un nom suffisamment explicite et différent des mots réservés de SQL et PL/SQL.
Toute variable doit avoir un type. Les types scalaires disponibles en PL/SQL sont :

- un des types prédéfinis d'ORACLE, i.e. ceux utilisés pour définir les attributs (e.g. **VARCHAR2**(n), **NUMBER**(x,y), **DATE**...) ;
- le type **BOOLEAN** qui admet les valeurs **TRUE, FALSE** et… **NULL** (Cf. paragraphe 3.2).

2.1.1 - Déclaration de variables

Outre son nom et son type, la déclaration d'une variable scalaire peut inclure l'affectation d'une valeur par défaut (évidemment compatible avec le type spécifié) et la clause **NOT NULL**. La syntaxe générale d'une déclaration de variable est la suivante :

```
      <nom_variable> <nom_type> [NOT NULL] [DEFAULT <valeur_defaut>] ;
ou    <nom_variable> <nom_type> [NOT NULL] [:=<valeur_defaut>] ;
```

La spécification d'une valeur par défaut peut se faire en utilisant soit le mot clef **DEFAULT** soit le symbole de l'affectation « := ». La valeur <valeur_defaut> peut être une valeur donnée explicitement ou résulter de l'utilisation d'une fonction ou d'un calcul.

Si la clause **NOT NULL** est utilisée, il faut obligatoirement donner une valeur par défaut à la variable concernée. En effet, si aucune valeur par défaut n'est indiquée, une valeur **NULL** est

utilisée pour initialiser la variable et une erreur est générée.

Exemple *: les déclarations suivantes sont valides en PL/SQL.*

```
effectif_groupe     NUMBER(2,0) ;
nombre_etudiants    NUMBER(4,0) := 0 ;
date_controle       DATE NOT NULL := sysdate ;
numero_ordre        NUMBER(2,0) DEFAULT 0 ;

/* Dans la déclaration suivante, une valeur par défaut est attribuée en
utilisant le contenu d'une variable préalablement déclarée */
date_ctl            DATE := date_controle + 30 ;
```

□

2.1.2 - Déclaration basée ou par référence

Il est possible lors de la déclaration d'une variable scalaire de ne pas indiquer explicitement un type mais de faire référence au type d'une autre variable, préalablement déclarée, ou au type d'un attribut d'une relation de la base. La variable dont le type est référencé peut elle-même être déclarée en faisant référence au type d'une autre variable ou d'un attribut.
La syntaxe de la déclaration devient alors :

```
    <nom_variable> <nom_reference>%TYPE [NOT NULL] [DEFAULT <valeur_defaut>];
ou  <nom_variable> <nom_reference>%TYPE [NOT NULL] [:=<valeur_defaut>] ;
```

`<nom_reference>` correspond au nom d'une autre variable ou d'un attribut de la base préfixé par le nom de la relation à laquelle il appartient : `<nom_relation>.<nom_attribut>`.

Exemple *: les déclarations suivantes sont valides en PL/SQL, en supposant que la variable* `effectif_groupe` *a déjà été déclarée.*

```
numero_etudiant     ETUDIANT.NUM_ET%TYPE ;
num_etudiant        numero_etudiant%TYPE ;
effectif_module     effectif_groupe%TYPE ;
effectif_projet     effectif_module%TYPE ;
```

□

Ce type de déclaration doit être utilisé chaque fois que c'est possible et préféré aux déclarations de type « en dur » car ce mécanisme est un facteur important pour minimiser la maintenance des applications. Plus précisément, il s'agit :

- ***d'augmenter l'indépendance entre données et programmes***. Si le type d'un attribut est modifié dans la base, il y aura peu, voire pas, de répercussions sur les programmes d'applications. Dans le meilleur des cas (par exemple extension de la taille d'un attribut), il suffit de recompiler le programme sans modification de code ;
- ***d'harmoniser le typage des variables***. Si différentes variables doivent être comparées dans les programmes, autant qu'elles partagent exactement le même type pour éviter les incompatibilités.

Si une variable est déclarée avec la clause **NOT NULL** et si elle est référencée dans la déclaration d'autres variables alors il faut que ces dernières soient déclarées avec une valeur par défaut. En effet ***la contrainte NOT NULL de la première variable est propagée aux variables la référençant***. Pour ces dernières, l'oubli de valeurs par défaut génère des erreurs de compilation puisqu'elles sont initialisées à **NULL**.

***Exemple** : une variable a été déclarée comme suit :*

```
numero_ordre      NUMBER(2,0) NOT NULL DEFAULT 0 ;
```

La déclaration suivante est invalide par manque d'une valeur par défaut.

```
numero_suivant    numero_ordre%TYPE ;
```

Mais les déclarations indiquées ci-après sont correctes.

```
numero_initial    numero_ordre%TYPE DEFAULT 1 ;
numero_final      numero_ordre%TYPE := 99 ;
```

□

2.2 - Les constantes

Les constantes sont déclarées d'une manière similaire aux variables scalaires, mais le mot clef **CONSTANT** est spécifié avant leur type. La valeur de la constante doit être indiquée soit explicitement soit comme résultat d'une fonction ou d'un calcul. Bien que les deux possibilités soient offertes, préférez le symbole := plutôt que **DEFAULT** car il s'agit d'une valeur permanente et non d'une valeur par défaut. La syntaxe à utiliser est donnée ci-après.

```
<nom_constante>   CONSTANT   <nom_type>  := valeur ;
```

***Exemple** : les déclarations suivantes sont valides en PL/SQL.*

```
effectif_max      CONSTANT NUMBER(2,0) := 30 ;
nom_diplome       CONSTANT CHAR(3) := 'DUT' ;
```

□

2.3 - Les variables composées

Les variables composées peuvent être de deux types : enregistrement (record) ou table. Dans le premier cas, la variable est constituée de champs qui peuvent être des variables scalaires ou composées. Dans le second cas, la variable est une collection d'éléments de même nature dont le type est forcément scalaire. En d'autres termes, les variables PL/SQL de type table sont des tableaux à une dimension d'éléments simples.

2.3.1 - Déclaration de variables enregistrements

Lorsqu'un développeur veut utiliser des variables enregistrements, il doit au préalable définir un type **RECORD** puis déclarer les variables de ce type. La syntaxe de ces ordres est la suivante.

```
TYPE <nom_type_enreg> IS RECORD
     (<nom_champ1> <nom_type1> [NOT NULL] [:= <valeur_par_defaut>],
     [<nom_champ2> <nom_type2> [NOT NULL] [:= <valeur_par_defaut>],… ]) ;

<nom_variable_enreg> <nom_type_enreg> ;
```

Les types spécifiés pour les différents champs sont les types scalaires vus précédemment, mais il est aussi possible de déclarer un champ en référençant le type d'un autre attribut ou variable. Enfin, les types enregistrements déclarés par l'utilisateur peuvent également être utilisés. L'interdiction de valeur nulle et l'attribution de valeur par défaut ne sont possibles que pour les

champs d'un type scalaire.

Exemple *: le développeur définit un type* ***RECORD*** *permettant de manipuler des variables composées « contrôles », décrites par un identifiant, une date, un type, des notes minimale et maximale ainsi que la matière contrôlée.*

```
TYPE controle_matiere IS RECORD(
  id_controle NUMBER(2,0),
  date_controle DATE,
  type_controle VARCHAR2(15),
  note_min NOTATION.MOY_TEST%TYPE := 0,
  note_max NOTATION.MOY_TEST%TYPE := 20,
  matiere_ctrlee MODULE.CODE%TYPE);
```

Il déclare ensuite des variables du type défini.

```
controle_bd controle_matiere ;
controle_sys controle_matiere ;
```

□

Lorsqu'un programme manipule des enregistrements, il est possible de traiter les champs de ces enregistrements dans le code PL/SQL en donnant le nom du champ préfixé par le nom de la variable enregistrement.

Exemple *: En supposant que le type et les variables de l'exemple précédent aient été déclarés, les expressions suivantes sont valides.*

```
controle_bd.date_controle < sysdate
controle_bd.note_min < controle_sys.note_min
```

□

Il est possible d'imbriquer les types **RECORD** en déclarant les champs d'un type enregistrement comme étant eux-mêmes d'un type **RECORD**. Pour désigner les champs imbriqués, la notation pointée est utilisée pour former un « chemin d'imbrication ».

```
TYPE <type_enreg1> IS RECORD
     (<nom_champ1> <type_scal_1> [NOT NULL] [:= <valeur_par_defaut>],
     [<nom_champ2> <type_scal_2> [NOT NULL] [:= <valeur_par_defaut>],…]);

TYPE <type_enreg2> IS RECORD
     (<nom_champ3> <type_enreg1>,
     [<nom_champ4> <type_scal_1> [NOT NULL] [:= <valeur_par_defaut>],…]);
```

Si la variable <nom_variable_enreg> est déclarée du type <type_enreg2>, ses champs sont désignés par :

```
<nom_variable_enreg>.<nom_champ4>
<nom_variable_enreg>.<nom_champ3>
<nom_variable_enreg>.<nom_champ3>.<nom_champ1>
<nom_variable_enreg>.<nom_champ3>.<nom_champ2>
```

Exemple *: pour manipuler des adresses et des numéros de téléphone, les types composés suivants sont définis à partir de champs scalaires. Un autre type est alors introduit pour gérer des coordonnées. Ses champs sont déclarés comme étant des types* ***RECORD*** *précédents.*

```
TYPE adresse IS RECORD(
  ligne1_adr VARCHAR2(40),
  ligne2_adr VARCHAR2(40),
```

```
    code_postal NUMBER(5, 0),
    ville VARCHAR2(30),
    pays VARCHAR2(30));

  TYPE telephone IS RECORD(
    ind_pays VARCHAR2(5),
    ind_region VARCHAR2(5),
    num_tel VARCHAR2(12));

  TYPE coordonnees IS RECORD(
    adresse_post adresse,
    numero_fixe telephone,
    numero_mobile telephone,
    mail VARCHAR2(30));

  coord_etud coordonnees;
```

La variable ci-dessus est déclarée en utilisant le type composé imbriqué `coordonnees`*. Ses champs sont accessibles de la manière suivante.*

```
coord_etud.numero_fixe              -- numéro complet du téléphone fixe
coord_etud.adresse_post             -- adresse complète
coord_etud.numero_fixe.ind_pays     -- indicatif du pays pour le numéro
                                    -- du téléphone fixe
coord_etud.adresse_post.ville       -- ville dans l'adresse postale
coord_etud.adresse_post.code_postal -- code postal de l'adresse postale□
```

2.3.2 - Déclaration basée ou par référence de variables enregistrements

Il existe une correspondance naturelle entre le concept d'enregistrement non imbriqué et celui de tuple. Vu l'intérêt pour la maintenance des applications des déclarations par référence, il est normal que PL/SQL les autorise pour les variables enregistrements. Ainsi, de la même manière que pour les variables scalaires, la déclaration de variables enregistrements peut faire référence à la structure des tuples d'une relation, d'une vue ou des tuples stockés dans un curseur (i.e. résultats d'une requête sur la base, Cf. paragraphe 6). Le mot clef utilisé, pour déclarer une telle référence, est **`%ROWTYPE`**.

Chaque fois qu'un programme manipule, de manière procédurale, des tuples issus d'une relation de la base, il faut utiliser une déclaration d'enregistrement par référence afin de minimiser la maintenance ultérieure de l'application. En effet, si la structure de la relation évolue par l'ajout, la suppression d'attributs ou la modification de leur type, il sera inutile de modifier la définition d'une variable déclarée par référence à cette relation.

Si les tuples devant être manipulés sont le résultat d'une requête effectuant la projection d'une relation ou un partitionnement avec fonction agrégative, ou encore si ce résultat est issu de plusieurs relations, la définition d'un curseur est nécessaire (Cf. paragraphe 6). Une déclaration par référence d'une variable enregistrement se fait de la même manière qu'elle soit basée sur la structure d'une table ou d'un curseur :

```
<nom_variable_enreg> <nom_table|nom_curseur>%ROWTYPE ;
```

Outre le fait que l'enregistrement peut être traité comme un tout en spécifiant simplement le nom de la variable dans le code PL/SQL, il est possible de manipuler chacune des valeurs le

composant en préfixant le nom de l'attribut sous-jacent par le nom de la variable enregistrement.

Exemple *: supposons que l'on veuille manipuler dans un programme les matières enseignées. Une variable enregistrement est déclarée comme ayant la même structure que les tuples de la relation MODULE.*

```
matiere_courante MODULE%ROWTYPE ;
```

Si le développeur veut ensuite utiliser le code et le libellé de la matière stockée dans cette variable, il le fait simplement par : `matiere_courante.CODE` *et* `matiere_courante.LIBELLE` *i.e. en utilisant le nom des attributs de la relation MODULE préfixé par le nom de la variable.* □

2.3.3 - Déclaration de variables tableaux

À partir de la version 2 de PL/SQL, il est possible d'utiliser des variables tableaux en définissant des types tables. En PL/SQL, une table est un vecteur d'éléments de même nature qui sont indexés par des entiers. La syntaxe est la suivante.

```
TYPE <nom_type_tableau> IS TABLE OF <type_element> [NOT NULL]
INDEX BY BINARY_INTEGER ;
```

où `<nom_type_tableau>` est le nom du type respectant les contraintes des identificateurs PL/SQL et `<type_element>` indique le type des éléments du tableau qui est soit un type scalaire de PL/SQL soit une référence à un autre type scalaire utilisant **`%TYPE`**.

La clause optionnelle **`NOT NULL`** permet d'imposer que tout élément ***créé*** ait une valeur. Enfin, la spécification de l'index de la table est obligatoirement **`INDEX BY BINARY_INTEGER`**. Codé en binaire, ce type d'index est en effet le plus rapide pour accéder aux éléments d'un tableau.
La déclaration d'une variable de type tableau est de la forme :

```
<nom_variable_tableau> <nom_type_tableau> ;
```

Exemple *: les déclarations de types et variables suivantes sont valides.*

```
TYPE table_deadlines IS TABLE OF DATE INDEX BY BINARY_INTEGER ;

TYPE table_notes IS TABLE OF NOTATION.MOY_TEST%TYPE
INDEX BY BINARY_INTEGER ;

les_deadlines table_deadlines ;
notes_test table_notes ;
```
□

Par l'utilisation de **`BINARY_INTEGER`**, les valeurs d'indice utilisées pour les variables tableaux varient dans l'intervalle de $[-2^{31} - 1 .. 2^{31} - 1]$. Pour manipuler les éléments d'une variable tableau, il faut indiquer le nom de la variable et, entre parenthèses, la valeur d'indice associée à l'élément considéré.
Il faut remarquer que les variables tableaux sont ***non bornées*** : aucune taille maximale n'est indiquée. Simplement les éléments sont ajoutés au fur et à mesure de leur création. De plus, les tableaux sont ***non denses***, les valeurs d'indice des éléments créés peuvent très bien ne pas être consécutives.

Exemple *: en supposant que les types et variables de l'exemple précédent aient été définis,* `les_deadlines(10)` *permet de manipuler la date stockée dans la table avec la valeur d'indice 10, si elle existe.* □

Plusieurs primitives, permettant de manipuler des variables tableaux, sont proposées :

- `<nom_variable_tab>.`**`EXISTS`**`(n)` rend la valeur `true` si il existe un élément d'indice `n` dans le tableau et `false` sinon ;
- `<nom_variable_tab>.`**`COUNT`** retourne le nombre d'éléments dans le tableau. Si ce dernier est vide, la valeur rendue est 0 ;
- `<nom_variable_tab>.`**`FIRST`** et `<nom_variable_tab>.`**`LAST`** rendent respectivement la valeur du plus petit indice dans le tableau et la valeur du plus grand. Si le tableau est vide, la valeur retournée dans les deux cas est **`NULL`** ;
- `<nom_variable_tab>.`**`PRIOR`**`(n)` et `<nom_variable_tab>.`**`NEXT`**`(n)` restituent l'indice de l'élément précédant ou suivant l'élément d'indice `n`. Si un tel élément n'existe pas, **`NULL`** est retourné.

2.4 - Les exceptions

Les exceptions, pouvant survenir lors de l'exécution d'un programme PL/SQL, peuvent être des exceptions système ou des exceptions développeur. ***Les premières sont automatiquement déclenchées par le SGBD lorsqu'il détecte une erreur*** (e.g. violation d'une contrainte de clef primaire, tentative de division par zéro…). Les exceptions développeur sont celles prévues par le programmeur et correspondent aux anomalies pouvant survenir pendant le fonctionnement de l'application. Seules les exceptions développeur doivent être définies dans la partie déclarative d'un bloc PL/SQL, en utilisant la syntaxe indiquée ci-après.

```
nom_exception   EXCEPTION ;
```

Qu'elles soient système ou utilisateur, toutes les exceptions doivent être traitées par le développeur (le code de traitement apparaît dans la partie `<gestion_exceptions>` du bloc, Cf. paragraphe 4). Un complément sur les exceptions systèmes est donné au paragraphe 4.

Exemple *: les exceptions utilisateur suivantes sont spécifiées dans la partie déclarative d'un programme.*

```
aucun_etudiant EXCEPTION ;
mauvais_argument EXCEPTION ;
```
□

3 - Instructions PL/SQL

Hormis l'instruction CASE, PL/SQL offre différentes instructions et les structures de contrôle classiques : contrôles conditionnel, itératif et séquentiel.

3.1 - Affectation

Il existe deux possibilités pour réaliser des affectations en PL/SQL. La première est classique et présente dans tout langage de programmation. La seconde est originale puisque l'affectation est réalisée via une requête sur la base de données.

3.1.1 - Affectation classique

Le symbole de l'affectation est := et cette opération est utilisée pour modifier le contenu d'une variable scalaire, celui d'un champ dans une variable enregistrement ou celui d'un élément dans un tableau, en utilisant la syntaxe suivante.

```
<nom_variable_scalaire> := <expression> ;
<nom_variable_enreg>.<nom_attribut> := <expression> ;
<nom_variable_enreg>.<nom_champ> := <expression> ;
<nom_variable_enreg>.<nom_champ_comp>.<nom_champ_scal> := <expression> ;
<nom_variable_table>(<indice>) := <expression> ;
```

où <expression> est soit une valeur compatible avec la variable, le champ d'un enregistrement ou l'élément d'une table, soit le contenu d'une autre variable ou d'une constante (évidemment compatible), soit le résultat d'une fonction ou d'un calcul (respectant la même contrainte).

Exemple *: supposons que les types et variables suivants aient été déclarés.*

```
TYPE controle_matiere IS RECORD (
  id_controle NUMBER(2,0),
  date_controle DATE,
  type_controle VARCHAR2(15),
  note_min NOTATION.MOY_TEST%TYPE := 0,
  note_max NOTATION.MOY_TEST%TYPE := 20
  matiere MODULE.CODE%TYPE);

TYPE table_notes IS TABLE OF NOTATION.MOY_TEST%TYPE
INDEX BY BINARY_INTEGER ;

num_et ETUDIANT.NUM_ET%TYPE ;
numero ETUDIANT.NUM_ET%TYPE DEFAULT 0;
maximum NOTATION.MOY_TEST%TYPE DEFAULT 20;
etudiant_courant ETUDIANT%ROWTYPE ;
controle_bd controle_matiere ;
notes_test table_notes ;
liste_notes table_notes ;
```

Les affectations suivantes sont valides :

```
num_et := 100 ;
num_et := num_et + 1 ;
num_et := least(100, numero) ;
etudiant_courant.ADR_ET := 'MARSEILLE' ;
controle_bd.note_max := 16 ;
notes_test(4):= maximum ;
notes_test(1):= liste_notes(4) ;
notes_test(10):= greatest(controle_bd.note_max, maximum) ;
```

□

L'affectation classique peut également opérer de manière groupée sur des variables enregistrements ou tableaux, mais elle a certaines limites. Plus précisément, les variables composées doivent non seulement avoir la même structure mais aussi être issues d'une même source. Ainsi si deux types différents sont déclarés avec exactement la même définition de champs (structures identiques), les variables de ces deux types ne sont pas comparables (sources

différentes).

Exemple *: supposons que les types* `controle_matiere` *et* `table_notes` *de l'exemple précédent aient été déclarés. Les variables suivantes sont définies.*

```
etudiant_courant ETUDIANT%ROWTYPE ;
etudiant_precedent ETUDIANT%ROWTYPE ;
controle_bd controle_matiere ;
controle_sys controle_matiere ;
notes_test1 table_notes ;
notes_test2 table_notes ;
```

Les affectations suivantes sont valides :

```
etudiant_courant := etudiant_precedent ;
controle_bd := controle_sys ;
notes_test1 := notes_test2 ;
```

Supposons que le type `controle` *soit déclaré comme un type enregistrement avec les mêmes champs que* `controle_matiere`*. Il n'est pas possible d'affecter le contenu d'une variable du type* `controle` *à une variable du type* `controle_matiere`*.* □

3.1.2 - Affectation par extraction de données de la base

Il est possible d'extraire ***une*** valeur ou ***un*** tuple de la base de données pour l'affecter soit à une variable scalaire soit à une variable enregistrement ou plusieurs variables scalaires. Pour cela, une requête doit être formulée avec la syntaxe suivante :

```
SELECT <nom_attribut | liste_attributs | *>
INTO <variable_receptrice>
FROM ... WHERE ...
```

La requête peut être aussi complexe que possible (inclure des blocs imbriqués, des **`GROUP BY`**, **`CONNECT BY`**…), du moment qu'elle ne rend qu'***au plus un résultat***. La clause **`SELECT`** de cette requête peut prendre plusieurs formes en fonction de la nature de la variable réceptrice :

- si `<variable_receptrice>` est une variable scalaire alors un seul nom d'attribut doit apparaître dans le **`SELECT`** ou une seule expression de calcul ou une fonction agrégative ;
- si `<variable_receptrice>` est une liste de variables scalaires alors une liste de noms d'attributs ou de calculs doit apparaître dans le **`SELECT`** en respectant le même ordre (les différents éléments doivent être comparables deux à deux). Le symbole `*` peut être utilisé ;
- si `<variable_receptrice>` est une variable enregistrement alors une liste de noms d'attributs ou d'expressions de calcul doit apparaître dans le **`SELECT`** en respectant l'ordre des champs de l'enregistrement. Le symbole `*` peut être utilisé.

Dans tous les cas, le développeur doit s'assurer que le résultat est logiquement constitué au plus d'un élément (i.e. que cela ne résulte pas d'une instance particulière de la base).

Exemple *: Le développeur veut manipuler dans son programme le nombre d'étudiants stockés dans la BD. La variable* `nb_etudiant` *est déclarée puis initialisée comme suit.*

```
nb_etudiant NUMBER(2,0);
```

```
SELECT COUNT(*) INTO nb_etudiant FROM ETUDIANT ;
```
□

Exemple *: On souhaite disposer, dans deux variables, de l'effectif des étudiants notés et de la moyenne de leur note de tests dans la matière de code BD. Les variables utilisées sont déclarées puis initialisées de la manière suivante :*

```
effectif NUMBER(3,0) ;
moyenne_bd NOTATION.MOY_TEST%TYPE ;

SELECT COUNT(NUM_ET), AVG(MOY_TEST) INTO effectif, moyenne_bd
FROM NOTATION WHERE CODE = 'BD' ;
```
□

Exemple *: Un programme doit manipuler un tuple de la relation ETUDIANT. La variable suivante est déclarée.*

```
un_etudiant ETUDIANT%ROWTYPE ;
```

L'affectation suivante est valide car la requête retourne au plus un tuple. S'il n'existe pas d'étudiant de numéro 2224, une exception système est déclenchée.

```
SELECT * INTO un_etudiant
FROM ETUDIANT WHERE NUM_ET = 2224 ;
```
□

Dans tous les cas de figure, il faut s'assurer que ***la requête ne retourne qu'au plus un résultat*** qui peut être une simple valeur ou un tuple de valeurs. Si cette condition n'est pas respectée, une erreur survient. Donc chaque fois que vous ne pouvez pas garantir l'unicité du résultat ***à tout instant de la vie d'une base***, il ne faut pas utiliser ce type d'affectation mais définir un curseur (Cf. paragraphe 6).

Lorsque la requête d'affectation ne rend aucun résultat, une exception système **NO_DATA_FOUND** est levée (Cf. paragraphe 4). Il faut remarquer le cas particulier des fonctions agrégatives : même si aucun tuple n'est retourné par la requête, l'exception **NO_DATA_FOUND** n'est pas levée. De plus le résultat de la fonction **COUNT** est 0 et celui des autres fonctions agrégative est **NULL**.
Les constantes de sélection d'une requête SQL incluse dans un bloc PL/SQL peuvent être données seulement au moment de l'exécution (comme en SQL*PLUS). Leur nom doit alors débuter par &.

3.2 - Instruction conditionnelle

En PL/SQL, les instructions conditionnelles ont la syntaxe suivante.

```
IF  <condition1>
THEN  [BEGIN]
      <liste1_instructions>
      [END ;]
[ELSIF <condition2>
THEN  [BEGIN]
       <liste2_instructions>
      [END ;] ]
[ELSE  [BEGIN]
       <liste3_instructions>
      [END ;]]
END IF ;
```

Les conditions peuvent utiliser des variables booléennes, des expressions utilisant les opérateurs de comparaison classiques (<, >, =, <> …), ainsi que **IS NULL** et **IS NOT NULL**. En fait tous les prédicats conditionnels disponibles en SQL*PLUS peuvent être utilisés. Ainsi les expressions suivantes sont des conditions valides en PL/SQL :

- <nom_variable> **IS** [**NOT**] **NULL**
- <nom_variable> [**NOT**] **BETWEEN** <borne_inf> **AND** <borne_sup>
- <nom_variable> [**NOT**] **IN** <liste_valeurs>
- <nom_variable> [**NOT**] **LIKE** <chaine_generique>

Les conditions simples peuvent être liées par **AND** et **OR**. **NOT** peut être utilisé pour des conditions négatives. Les mots clefs **BEGIN** et **END** ne sont nécessaires que si <liste_instructions> comporte effectivement plusieurs instructions (chacune terminée par ;).
Enfin, il est possible d'imbriquer des instructions **IF**.

Exemple *: on suppose que les variables utilisées ont été préalablement déclarées et initialisées. Le premier exemple rappelle la syntaxe d'un simple* ***IF-THEN****.*

```
IF  note > moyenne_notes
THEN
      nb_bons_etudiants := nb_bons_etudiants + 1 ;
END IF ;
```

*Le code indiqué ci-après montre la gestion de conditions exclusives (****IF-ELSIF****).*

```
IF  note > moyenne_notes
THEN  BEGIN
      nb_etudiants_bons := nb_etudiants_bons + 1 ;
      appreciation := 'BON';
      END ;
ELSIF note > moyenne_notes / 2
THEN  BEGIN
      nb_etudiants_moyens := nb_etudiants_moyens + 1 ;
      appreciation := 'NEUTRE';
      END ;
ELSE  BEGIN
      nb_etudiants_faibles := nb_etudiants_faibles + 1 ;
      appreciation := 'FAIBLE' ;
      END ;
END IF ;
```

Le code indiqué ci-après illustre l'imbrication de ***IF****. On suppose que la variable* `redoublant` *est de type booléen et que les procédures* `alerte` *et* `editer_convocation` *ont des paramètres, non précisés ici.*

```
IF  moyenne_generale < 10
THEN  BEGIN
      IF redoublant
      THEN
            alerte(…) ;
      END IF ;
      editer_convocation(…) ;
      END ;
END IF ;
```

□

Lors de l'évaluation de conditions, il convient de faire attention aux variables susceptibles de

contenir des valeurs nulles. En général toute comparaison dont un terme est **NULL** est évaluée à **NULL** et la comparaison échoue. Il faut toujours se rappeler que **NULL** n'est jamais égal à une valeur effective et est toujours différent d'une valeur effective [1]. Généralement appliquer une fonction de calcul horizontal à une variable ayant la valeur **NULL** rend… **NULL**. Appliquer l'opérateur **NOT** à une expression évaluée à **NULL** rend **NULL** [5]. Les prédicats **IS NULL** et **IS NOT NULL** peuvent être utilisés dans les tests.

Exemple *: dans le test suivant, si la variable* `nom_etudiant` *est à* **NULL**, *le résultat de l'évaluation est* **NULL**.

```
DECLARE
   nom_etudiant ETUDIANT.NOM_ET%TYPE;
BEGIN
…
   IF length(nom_etudiant) = 0
   THEN
      traiter_etudiant_inexistant(…) ;
   END IF ;
END ;
```

Si le souhait du développeur est de traiter les étudiants dépourvus de nom, le test donné va systématiquement écarter ce cas car l'expression **length**`(nom_etudiant)=0` *n'est pas évaluée à* **TRUE**. *La condition doit être remplacée par :* `nom_etudiant` **IS NULL** *ou* **length** `(nom_etudiant)` **IS NULL**. □

3.3 - Itérations

Les instructions d'itération peuvent être des boucles simples (**LOOP**), des boucles **FOR**[3] ou des boucles **WHILE**. Il est évidemment possible de les imbriquer.
Les trois types de boucles peuvent être utilisées pour parcourir les éléments d'un tableau en utilisant les primitives données au paragraphe 2.3.3.

3.3.1 - Boucle simple

La syntaxe d'une boucle simple est donnée ci-dessous.

```
[<<etiquette_boucle>>]
LOOP
    <liste_instructions>
END LOOP [etiquette_boucle] ;
```

Les boucles simples doivent être utilisées avec beaucoup de soins. En effet, ces boucles ne s'arrêtent que si les ordres **EXIT** ou **EXIT WHEN** `<condition>` sont rencontrés lors de l'exécution, donc attention aux boucles infinies. Ces boucles servent à simuler des boucles de type REPETER-JUSQU'A et permettent d'entrer au moins une fois dans la boucle spécifiée. Pour une meilleure lisibilité, les boucles peuvent être nommées par : `<<etiquette_boucle>>`. Dans ce cas, le nom attribué doit être mentionné après **END LOOP**.

[3] Seules sont considérées dans ce paragraphe les boucles **FOR** numériques. Un autre type de boucle **FOR** est présenté pour la gestion des curseurs (Cf. paragraphe 6).

Exemple *: dans la boucle simple suivante, on suppose que les variables utilisées ont été préalablement déclarées et que la procédure* `affecter_groupe` *est définie. De plus la variable* `maximum` *est correctement initialisée.*

```
numero := 1;
LOOP
       affecter_groupe(…) ;
       numero := numero + 1 ;
       EXIT WHEN numero > maximum ;
END LOOP ;
```
□

Exemple *: la variable de type tableau* `liste_notes` *a été déclarée comme une table PL/SQL dont les éléments sont du même type que NOTATION.MOY_TEST et ses éléments ont été initialisés. La boucle simple suivante permet de parcourir ce tableau et déclenche la procédure* `agreger_notes` *à chaque nouvelle itération. La sortie de la boucle est effectuée lorsque aucune valeur n'est retournée par* ***NEXT****(i).*

```
i := liste_notes.FIRST ;
<<boucle_agregation>>
LOOP
       agreger_notes(…) ;
       i := liste_notes.NEXT(i) ;
       EXIT WHEN i IS NULL;
END LOOP boucle_agregation ;
```
□

3.3.2 - Boucle FOR

Les boucles **FOR** numériques sont spécifiées de la manière suivante.

```
[<<etiquette_boucle>>]
FOR <compteur> IN [REVERSE] <borne_inf>..<borne_sup>
LOOP
    <liste_instructions>
END LOOP [etiquette_boucle] ;
```

La variable de boucle `<compteur>` ne doit pas être déclarée car elle est automatiquement générée par le système. Une déclaration supplémentaire de la part du développeur engendre une ambiguïté et peut être source d'erreurs difficiles à détecter. Si le mot clef **REVERSE** n'est pas précisé, la variable de boucle est initialisée à la valeur `<borne_inf>` et incrémentée de 1 à chaque nouvelle itération jusqu'à atteindre `<borne_sup>`. Les valeurs des deux bornes doivent toujours être spécifiées dans l'ordre indiqué même si les valeurs initiale et finale sont inversées par l'utilisation de **REVERSE**. Le pas de l'incrément est toujours 1.

Exemple *: la boucle indiquée déclenche une procédure autant de fois que la valeur de la variable* `maximum`*.*

```
FOR numero IN 1..maximum LOOP
       affecter_groupe(…) ;
END LOOP ;
```
□

Exemple *: le parcours d'un tableau est réalisé en utilisant les primitives* ***FIRST*** *et* ***LAST****.*

```
FOR i IN liste_notes.FIRST .. liste_notes.LAST LOOP
       agréger_note(…) ;
END LOOP ;
```
□

3.3.3 - Boucle WHILE

La dernière possibilité d'itération est la boucle **WHILE**.

```
[<<etiquette_boucle>>]
WHILE <condition>
LOOP
    <liste_instructions>
END LOOP [etiquette_boucle] ;
```

***Exemple** : la boucle indiquée déclenche une procédure autant de fois que la valeur de la variable `maximum`.*

```
numero := 1;
WHILE numero <= maximum LOOP
       affecter_groupe(…);
       numero := numero + 1 ;
END LOOP ;
```
□

***Exemple** : la boucle ci-dessous effectue le parcours des éléments d'un tableau.*

```
i := liste_notes.FIRST;
WHILE i IS NOT NULL LOOP
       agreger_note(…);
       i := liste_notes.NEXT(i);
END LOOP ;
```
□

3.4 - Autres instructions

PL/SQL permet la possibilité de branchement sur une autre partie du code via l'ordre **GOTO** dont l'argument est un nom d'étiquette. Dans le code, le nom de cette étiquette apparaît entre << et >>. Cette instruction peut être utile par exemple pour éviter de transférer le contrôle du programme à la partie traitement des exceptions. Il est interdit de réaliser un branchement sur des instructions d'un **IF**, d'une boucle ou d'un sous bloc. De plus, l'instruction **GOTO** ne peut être utilisée dans la partie gestion des exceptions.

Souvent utile, l'instruction **NULL** permet… de ne rien faire. Elle peut par exemple être utilisée, si une exception système est levée, pour que le programme se termine, malgré tout, normalement. En phase de développement, pour tester la validité de la partie déclarative, cette instruction peut être la seule du bloc.

***Exemple** : le corps d'un bloc PL/SQL intégrant ces instructions est donné ci-dessous.*

```
SELECT COUNT(*) INTO nb FROM ETUDIANT WHERE ANNEE = 2 ;
IF nb = 0 THEN GOTO terminaison ;
ELSE …
END IF ;
```

```
…
<<terminaison>>
     NULL ;
END ;
```
□

4 - Gestion des exceptions

En PL/SQL, les diverses erreurs qui peuvent survenir au cours de l'exécution d'un programme sont gérées comme des exceptions. Il existe deux catégories d'exception, les exceptions développeur et les exceptions système. Dans chaque cas, les exceptions peuvent être nommées ou anonymes. Les exceptions sont déclenchées lorsque l'anomalie correspondante est détectée au cours de l'exécution d'un programme PL/SQL. Le contrôle du programme est alors passé à la partie traitement des exceptions du bloc. Toute erreur ORACLE, détectée au cours de l'exécution et non traitée par le développeur, génère l'arrêt du traitement et le ***SGBD défait la transaction courante***.

	Exceptions Développeur	**Exceptions Système**
Nommées	• doivent être déclarées • doivent être déclenchées • doivent être gérées	• sont nommées par Oracle • sont déclenchées par Oracle • doivent être gérées par le développeur
Anonymes	• sont spécifiques des blocs stockés ou des triggers • sont déclenchées via `RAISE_APPLICATION_ERROR`	• sont identifiées par un numéro d'erreur • peuvent être nommées • sont déclenchées par Oracle • doivent être gérées par le développeur

Tableau 1 : Les différents types d'exceptions

4.1 - Les exceptions système nommées et anonymes

Les exceptions système sont automatiquement déclenchées lors d'une erreur ORACLE. Les plus courantes sont nommées, elles sont répertoriées dans le tableau 1. Les autres sont simplement numérotées, elles sont alors dites anonymes.
Il existe également une exception générique **`OTHERS`** qui permet au programmeur de récupérer toutes les exceptions système qui ne sont pas gérées spécifiquement dans son traitement PL/SQL.

Nom de l'exception	Signification
`CURSOR_ALREADY_OPEN`	Tentative d'ouverture d'un curseur déjà ouvert.
`DUP_VAL_ON_INDEX`	Tentative d'insertion d'une valeur de clef primaire existante (duplication impossible dans l'index primaire associé).
`INVALID_CURSOR`	Tentative de manipulation ou de fermeture d'un curseur non ouvert.
`INVALID_NUMBER`	Une valeur numérique est incorrecte.
`NO_DATA_FOUND`	Aucun résultat n'est retourné par une requête (en particulier

	lors de l'affectation par un **SELECT** … **INTO** …) ou tentative de manipulation d'un élément non créé dans une variable tableau PL/SQL.
STORAGE_ERROR	Problème de mémoire.
TIMEOUT_ON_RESSOURCE	Le délai maximum d'attente d'une ressource (verrouillée par un autre utilisateur ou transaction) a expiré.
TOO_MANY_ROWS	Un ordre **SELECT** … **INTO** … a retourné plusieurs résultats.
VALUE_ERROR	Une erreur de conversion, de troncature ou de borne est détectée.
ZERO_DIVIDE	Tentative de division par zéro.

Tableau 2 : Principales exceptions système nommées

Les exceptions système anonymes identifiées par un code d'erreur interne à ORACLE peuvent être nommées par le développeur en utilisant la pseudo-instruction ou pragma[4] **EXCEPTION_INIT**. Cette attribution de nom doit être définie de la manière suivante dans la partie déclarative.

```
<nom_exception> EXCEPTION ;
PRAGMA EXCEPTION_INIT(<nom_exception>, <code_erreur_oracle>) ;
```

L'exception doit être déclarée avant l'utilisation de **EXCEPTION_INIT** et le code de l'erreur est généralement négatif, sauf pour l'exception de code 100 dont le message est : « no data found ». Les fonctions prédéfinies **sqlcode** et **sqlerrm** peuvent être utilisées, elles rendent respectivement le code d'une exception et le message associé. Ces fonctions sont particulièrement intéressantes lorsque l'exception **OTHERS** est utilisée afin d'effectuer des traitements spécifiques. Lorsqu'une exception utilisateur est déclenchée, **sqlcode** rend +1 et **sqlerrm** le message correspondant. Enfin, si aucune exception n'est levée au cours de l'exécution d'un bloc, **sqlcode** rend 0 et dans **sqlerrm** le message est «normal, successful completion ».
Les fonctions **sqlcode** et **sqlerrm** ne peuvent pas être directement utilisées dans un ordre SQL, il faut utiliser des variables locales intermédiaires.

***Exemple** : lorsqu'une insertion de tuple dans une relation est incomplète par manque de valeurs, l'exception système anonyme -947 est déclenchée. Pour pouvoir traiter ultérieurement cette exception, les déclarations suivantes sont spécifiées.*

```
valeur_manquante EXCEPTION ;
PRAGMA EXCEPTION_INIT(valeur_manquante, -947) ;
```
□

***Exemple** : le code suivant permet le traitement d'exception système anonyme sans utiliser de pragma.*

```
EXCEPTION
       WHEN OTHERS THEN
       IF sqlcode = -947 THEN …          -- Traiter insertion invalide
       ELSIF sqlcode = -68 THEN …        -- Traiter valeur invalide
       ELSE …
```
□

[4] i.e. une directive au compilateur.

4.2 - Déclenchement des exceptions utilisateur nommées

Les exceptions nommées spécifiées par l'utilisateur doivent être déclarées (Cf. paragraphe 2.4). Elles sont explicitement déclenchées par le programme lorsqu'une condition est satisfaite en respectant la syntaxe suivante.

```
IF  <condition>
THEN
       RAISE <nom_exception> ;
END IF  ;
```

4.3 - Déclenchement des exceptions utilisateur anonymes

Le développeur peut décider de lever une exception anonyme si certaines conditions ne sont pas satisfaites au cours de l'exécution d'un **programme stocké ou d'un trigger** (Cf. paragraphes 7.2.2 et 7.3.2). Le déclenchement d'une telle exception interrompt le traitement et ORACLE défait la transaction courante.
Pour déclencher des exceptions anonymes, le développeur doit utiliser :

```
IF <condition> THEN
    RAISE_APPLICATION_ERROR(<code_excep_ut>, <message_erreur>)
END IF;
```

où :

- `<code_excep_ut>` est une valeur entière négative comprise entre -20999 et –20000 ;
- `<message_erreur>` est le texte du message à afficher donné entre apostrophes.

***Exemple** : dans le code suivant, une exception utilisateur anonyme peut être déclenchée.*

```
…
   effectif_bd NUMBER(2,0) ;

BEGIN
   SELECT COUNT(NUM_ET) INTO effectif_bd
   FROM NOTATION WHERE CODE = 'BD' ;

   IF effectif_bd = 0 THEN
     RAISE_APPLICATION_ERROR (-20001,'Aucune note') ;
   END IF ;
       …
END ;
```

□

4.4 - Traitement des exceptions

Qu'elles soient système ou utilisateur, les exceptions doivent être gérées dans la partie **EXCEPTION** du programme PL/SQL en respectant la syntaxe suivante.

```
WHEN <nom_exception>
THEN  [BEGIN]
      <liste_instructions>
      [END] ;
```

Il est possible dans la clause **WHEN** de combiner des noms d'exceptions avec **OR**.

***Exemple** : supposons que la partie déclarative intègre les variables et exceptions suivantes.*

```
DECLARE
   effectif_bd NUMBER(2,0) ;
   min_note_bd NOTATION.MOY_TEST%TYPE ;
   note_bd NOTATION%ROWTYPE ;
   alerte_bd EXCEPTION ;
   absence_note EXCEPTION ;

BEGIN
   SELECT COUNT(NUM_ET), MIN(MOY_TEST) INTO effectif_bd, min_note_bd
   FROM NOTATION WHERE CODE = 'BD' ;

   IF effectif_bd = 0 THEN RAISE absence_note ;
   ELSIF
      BEGIN
         /* L'exception système NO_DATA_FOUND est levée si la requête
         suivante ne rend pas de résultat */
         SELECT * INTO note_bd FROM NOTATION
         WHERE NUM_ET = &1 AND CODE = 'BD' ;
         IF (note_bd.MOY_TEST - min_note_bd) <  3
         THEN
            RAISE alerte_bd ;
         END IF ;
      END ;
   END IF ;
EXCEPTION
   WHEN absence_note OR NO_DATA_FOUND THEN NULL ;
   WHEN alerte_bd THEN
      BEGIN
         convoquer(…) ;
         …
      END ;
END ;
```

□

5 - Ordres SQL valides en PL/SQL

Les ordres SQL qui peuvent être utilisés dans un bloc PL/SQL sont ceux relevant de l'aspect manipulation de données du langage (LMD), i.e. les suivants :

- Toute requête **SELECT** … **INTO** … **FROM** … dans la liste des instructions et **SELECT** … **FROM** … dans la partie déclarative pour définir un curseur.

- Les opérations de mise à jour des données : **INSERT, UPDATE, DELETE**.

- ***Sauf pour les triggers***, les ordres de gestion de transaction : **COMMIT, ROLLBACK, SAVEPOINT**.
  - **COMMIT** et **ROLLBACK** permettent de définir une transaction. En cas d'incident, les ordres de mise à jour des données pourront alors être « défaits » par **ROLLBACK** via une exception.
  - **SAVEPOINT** <nom_point_sauve> et **ROLLBACK TO** <nom_point_sauve> peuvent être utilisés pour ne défaire qu'une partie de la transaction.

De plus, il est possible d'utiliser :

- toutes les fonctions de calcul horizontal proposées en SQL, aussi bien dans les ordres SQL intégrés dans le code PL/SQL que dans les traitements procéduraux de PL/SQL. Dans ce dernier cas, la seule exception est la fonction **`decode`**.
- les pseudo-colonnes peuvent être utilisées dans les ordres SQL intégrés.

Les ordres de manipulation des composants d'un schéma (i.e. l'aspect LDD de SQL), comme la création de relation ou l'ajout d'attributs, ne peuvent pas être directement écrits comme des instructions dans le corps d'un bloc PL/SQL. À partir de la version 2 de PL/SQL (ORACLE v7.0), cette restriction peut être contournée en ayant recours à SQL dynamique (Cf. paragraphe 10).

6 - Les curseurs

Le rôle des curseurs est d'établir la transition entre l'univers BD et celui des langages procéduraux classiques. Plus précisément, un curseur permet d'extraire un ***ensemble de tuples***, résultat d'une requête d'interrogation sur la base de données et de le manipuler de manière procédurale, i.e. en balayant un par un les tuples/enregistrements. Un curseur peut être vu comme une zone de travail nommée contenant des tuples qu'il est possible de manipuler. Il peut aussi être perçu comme une table temporaire dont le contenu n'est préservé que le temps d'une session de travail.

Un curseur est défini dans la partie déclarative d'un bloc PL/SQL (1). La requête associée, spécifiée et déclenchée lors de l'exécution du corps du bloc, est transmise au SGBD qui l'exécute sur la base (2) et alimente le curseur à partir des résultats obtenus (3). Le bloc PL/SQL peut alors accéder itérativement aux enregistrements du curseur et manipuler les valeurs lues (4) et effectuer toute opération de mise à jour (écriture) directement sur la base de données. Ces principes généraux sont illustrés par la figure 2.

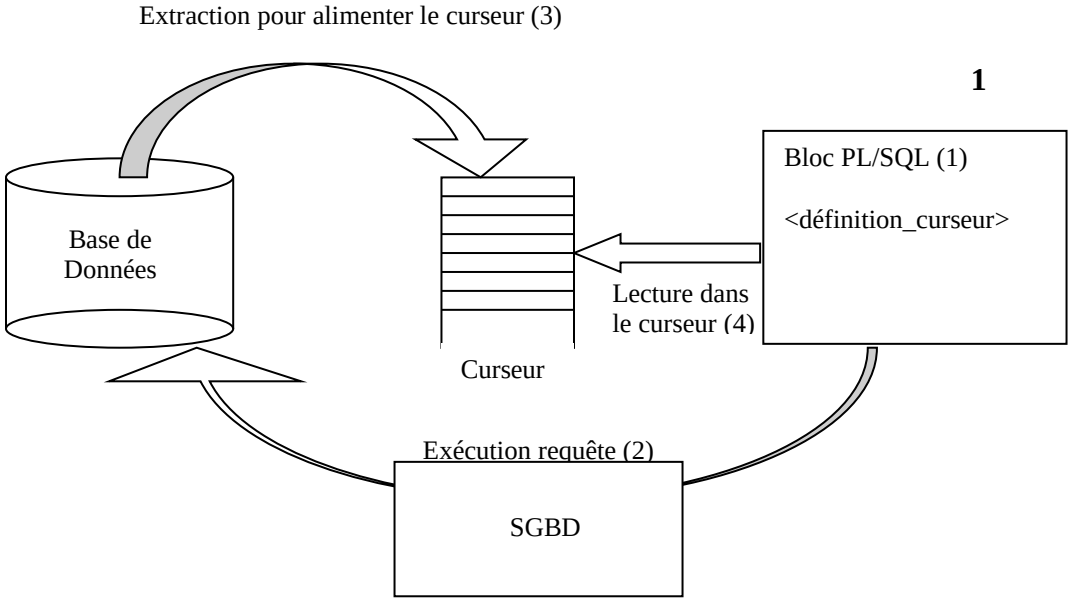

Figure 2 : Mécanisme des curseurs

6.1 - Définition de curseur

Un curseur est défini dans la partie déclarative d'un bloc PL/SQL par une requête d'interrogation en SQL (sa structure correspond aux attributs ou expressions mentionnés dans la clause **SELECT** de cette requête), en suivant la syntaxe suivante.

```
CURSOR <nom_curseur>  IS  <requete_SQL> ;
```

Exemple *: dans un programme, on souhaite manipuler la liste des étudiants. Le curseur suivant est déclaré.*

```
CURSOR C_etudiant  IS
       SELECT NUM_ET, NOM_ET, PRENOM_ET
       FROM ETUDIANT
       WHERE ANNEE = 2
       ORDER BY NOM_ET, PRENOM_ET;
```

Chaque enregistrement du curseur contiendra, après l'exécution de la requête de définition, un numéro, un nom et un prénom d'étudiant de deuxième année. □

Exemple *: la requête de définition d'un curseur peut travailler sur plusieurs relations de la base, ici sur MODULE et PROF.*

```
CURSOR C_matiere_resp  IS
       SELECT  CODE, LIBELLE, RESP, NOM_PROF  FROM  MODULE, PROF
       WHERE RESP = NUM_PROF
       ORDER  BY  CODE ;
```
□

Lorsqu'une expression de calcul horizontal ou une fonction agrégative est utilisée dans la requête de définition d'un curseur, il faut attribuer un alias à cette expression pour pouvoir ultérieurement manipuler son résultat. Cet alias peut être utilisé dans la clause **ORDER BY** de la requête.

Exemple *: la définition du curseur donnée ci-dessous intègre un calcul horizontal. L'alias* `MOYENNE` *est attribué à l'expression correspondante.*

```
CURSOR C_moyenne_matiere  IS
       SELECT CODE, NUM_ET,
       ( NVL(MOY_TEST,0) * COEFF_TEST + NVL(MOY_CC,0)* COEFF_CC )/
       ( COEFF_TEST+COEFF_CC ) MOYENNE
       FROM NOTATION
       ORDER  BY MOYENNE ;
```
□

Après la déclaration d'un curseur, il est possible de l'utiliser comme référence pour la déclaration de nouvelles variables enregistrements, exactement comme pour les relations de la base. Le nom des champs de la variable enregistrement ainsi déclarée sont les noms des attributs ou les alias spécifiés dans la requête de définition du curseur.

Exemple *: le curseur* `C_moyenne_matiere` *de l'exemple précédent est utilisé dans la*

déclaration d'une variable enregistrement.

```
moyenne_etudiant C_moyenne_matiere%ROWTYPE ;
```

Les champs de la variable enregistrement peuvent être manipulés en utilisant les expressions suivantes : `moyenne_etudiant.CODE, moyenne_etudiant.MOYENNE.` □

6.2 - Gestion de curseurs

La gestion de curseur peut se faire de manière manuelle ou automatique. Dans le premier cas, c'est au développeur qu'incombe le codage du parcours du curseur. Pour cela il existe trois ordres de manipulation d'un curseur permettant respectivement son ouverture, la récupération du prochain enregistrement dans une variable réceptrice et sa fermeture. Le principe de lecture cohérente d'ORACLE garantit que les données lues au moment de l'ouverture d'un curseur reflètent l'état de la base à cet instant là. Si des opérations de mise à jour ont lieu sur la base entre cet instant et la lecture d'enregistrement dans le curseur, ces mises à jour ne sont pas prises en compte dans le curseur.

- **OPEN** <nom_curseur> ;
  Lors de l'ouverture d'un curseur, PL/SQL associe la requête de définition au curseur et l'exécute mais à ce stade, le curseur n'a pas d'enregistrement courant.

- **CLOSE** <nom_curseur> ;
  La fermeture du curseur le désactive et libère la place mémoire occupée. Le curseur, n'ayant plus de contenu, ne peut plus être manipulé. Il faut systématiquement fermer les curseurs qui ne sont plus utilisés par le programme car ils peuvent occuper une place non négligeable dans la mémoire partagée globale du SGBD.

- **FETCH** <nom_curseur> **INTO** <variable-receptrice> ;
  retourne l'enregistrement courant du curseur et se positionne sur le prochain enregistrement. L'enregistrement récupéré est stocké dans <variable-receptrice> qui peut être une variable enregistrement de même structure que le curseur ou une liste de variables scalaires qui doivent être données en respectant l'ordre des attributs dans la requête de définition du curseur.

Exemple** : le bloc PL/SQL suivant détermine si l'unité d'enseignement, donnée par l'utilisateur lors de l'exécution, est un module (nœuds non terminaux dans la hiérarchie pédagogique). Le curseur* `Test_module` *extrait les fils d'un module. Si le code donné n'est pas celui d'un module, l'utilisation du curseur génère l'exception* ***NO_DATA_FOUND*. On suppose ici qu'on dispose d'une procédure* `afficher`.

```
DECLARE
CURSOR Test_module IS
     SELECT * FROM MODULE WHERE CODEPERE = '&module' ;

un_module Test_module%ROWTYPE ;          --variable réceptrice de
                                         --l'enregistrement courant
lib_module MODULE.LIBELLE%TYPE ;

BEGIN
     OPEN Test_module ;
     FETCH Test_module INTO un_module ;
```

```
        /* Quand la requête ci-dessous est exécutée, c'est qu'il existe un
        ou plusieurs enregistrements dans le curseur mais seul le premier
        est utilisé pour retrouver le libellé du module */

        SELECT LIBELLE INTO lib_module FROM MODULE
        WHERE CODE = un_module.CODEPERE ;
        afficher(lib_module||' est un module');
        CLOSE Test_module ;
    EXCEPTION
        WHEN NO_DATA_FOUND
        THEN afficher('Le code donné ne correspond pas à un module');
    END ;
```
□

6.3 - Propriétés des curseurs

Les propriétés (ou attributs) des curseurs donnent des informations sur l'état courant du curseur. Elles sont au nombre de quatre et correspondent à des propriétés booléennes ou numérique.

- `<nom_curseur>`**`%FOUND`**
  Cette propriété est à **`TRUE`** si le dernier ordre **`FETCH`** exécuté a retourné un enregistrement. Elle est à **`FALSE`** sinon. Après l'ouverture d'un curseur, si aucun ordre **`FETCH`** n'a été effectué, cette propriété est à **`NULL`**.

- `<nom_curseur>`**`%NOTFOUND`**
  Cette propriété, inverse de la précédente, est à **`TRUE`** si le dernier ordre **`FETCH`** exécuté ne peut pas retourner d'enregistrement supplémentaire. Elle est à **`FALSE`** sinon. Après l'ouverture d'un curseur, si aucun ordre **`FETCH`** n'a été effectué, cette propriété est à **`NULL`**.

- `<nom_curseur>`**`%ISOPEN`**
  Cette propriété est à **`TRUE`** si le curseur est ouvert et à **`FALSE`** s'il est fermé.

- `<nom_curseur>`**`%ROWCOUNT`**
  Cette propriété donne le nombre cumulé d'enregistrements ramenés par des ordres **`FETCH`** depuis que le curseur est ouvert. Elle vaut 0 à l'ouverture du curseur. Cette propriété est particulièrement intéressante quand il faut traiter un certain nombre d'enregistrements présents dans le curseur mais pas la totalité.

Ces propriétés peuvent être utilisées dans un programme PL/SQL en particulier pour éviter le déclenchement d'exceptions. L'utilisation de ces propriétés ne peut se faire que si le curseur est ouvert. Dans le cas contraire, une exception système **`INVALID_CURSOR`** est déclenchée.

***Exemple** : le curseur `C_moyenne_matiere` et la variable enregistrement `moyenne_etudiant` sont manipulés dans le code donné ci-après.*

```
DECLARE
    CURSOR C_moyenne_matiere  IS
           SELECT CODE, NUM_ET,
           ( NVL(MOY_TEST,0) * COEFF_TEST + NVL(MOY_CC,0)* COEFF_CC )/
           ( COEFF_TEST+COEFF_CC ) MOYENNE
           FROM NOTATION
           ORDER  BY MOYENNE ;
```

```
    moyenne_etudiant C_moyenne_matiere%ROWTYPE ;

BEGIN
    …
    /* Le contrôle suivant permet d'éviter le déclenchement de l'exception
    système CURSOR_ALREADY_OPEN */

    IF NOT C_moyenne_matiere%ISOPEN
    THEN
           OPEN C_moyenne_matiere ;
    END IF ;

    FETCH C_moyenne_matiere INTO moyenne_etudiant ;

    /* La caratéristique C_moyenne_matiere%FOUND est utilisée comme condition
    de boucle. Évidemment si le curseur est vide, aucune itération n'est
    effectuée */

    <<parcours_moyenne>>
    WHILE C_moyenne_matiere%FOUND
    LOOP
           IF moyenne_etudiant.CODE = 'BD'
           THEN
                 moyenne_etudiant.MOYENNE:= moyenne_etudiant.MOYENNE + 1 ;
           END IF ;
           FETCH C_moyenne_matiere INTO moyenne_etudiant ;
    END LOOP parcours_moyenne ;
```

Remarquons qu'un premier ordre ***FETCH*** *est nécessaire avant l'entrée dans la boucle pour pouvoir évaluer la propriété* `C_moyenne_matiere%FOUND`. □

Exemple *: reformulons l'exemple précédent avec une boucle de type* ***LOOP***. *Le premier ordre* ***FETCH*** *est intégré dans la boucle et est forcément exécuté. La sortie de la boucle se fait en utilisant un ordre* ***EXIT*** *dès qu'il n'y a plus d'enregistrement à traiter dans le curseur.*

```
    IF NOT C_moyenne_matiere%ISOPEN THEN OPEN C_moyenne_matiere ; END IF ;
    LOOP
           FETCH C_moyenne_matiere INTO moyenne_etudiant ;
           EXIT WHEN C_moyenne_matiere%NOTFOUND ;
           IF moyenne_etudiant.CODE = 'BD'
           THEN
                 moyenne_etudiant.MOYENNE:= moyenne_etudiant.MOYENNE + 1 ;
           END IF ;
    END LOOP ;
```

□

Exemple *: supposons qu'un traitement particulier soit effectué (ici appel de deux procédures) pour les étudiants ayant obtenu une des dix meilleures moyennes dans une matière. Le curseur suivant est utilisé.*

```
    CURSOR C_moyenne_matiere  IS
           SELECT CODE, NUM_ET,
           ( NVL(MOY_TEST,0) * COEFF_TEST + NVL(MOY_CC,0)* COEFF_CC )/
           ( COEFF_TEST+COEFF_CC ) MOYENNE
           FROM  NOTATION
           ORDER  BY  MOYENNE DESC ;
```

```
moyenne_courante NOTATION.MOY_TEST%TYPE;
```

Le code proposé est le suivant.

```
moyenne_courante := 0 ;
OPEN C_moyenne_matiere ;
LOOP
        FETCH C_moyenne_matiere INTO moyenne_etudiant ;
        EXIT WHEN C_moyenne_matiere%NOTFOUND ;
        EXIT WHEN C_moyenne_matiere%ROWCOUNT > 10 AND
              moyenne_etudiant.MOYENNE <> moyenne_courante ;
        moyenne_courante := moyenne_etudiant.MOYENNE ;
        attribuer_bonus(…) ;
        editer_lettre(…) ;
END LOOP ;
```

La propriété `%ROWCOUNT` *est utilisée pour garantir que le programme lit au moins les 10 premiers enregistrements (s'ils existent). Comme les enregistrements dans le curseur sont triés selon* `MOYENNE`*, les dix meilleures moyennes par matière sont traitées. La seconde condition utilisant la variable* `moyenne_courante` *permet de tenir compte des ex æquo.* □

6.4 - Parcours automatique de curseur

Chaque fois qu'un traitement concerne tous les enregistrements contenus dans un curseur, il est conseillé d'utiliser une boucle **FOR** de parcours automatique de curseur avec la syntaxe associée est donnée ci-après.

```
FOR <nom_variable_parcours> IN <nom_curseur>
LOOP
        <liste_instructions>
END LOOP ;
```

En utilisant ce type de boucle, il ne faut pas ouvrir le curseur, ni spécifier d'ordres **FETCH** ni enfin fermer le curseur. Tout cela est fait automatiquement. De plus la variable de parcours de boucle ne doit pas être déclarée pour éviter toute ambiguïté. Comme pour les variables enregistrements, l'accès aux champs de l'enregistrement courant se fait en préfixant le nom du champ par le nom de la variable de parcours.

Exemple *: supposons que le curseur* `C_moyenne_matiere` *soit déclaré comme dans les exemples précédents. Augmenter les moyennes dans la matière BD peut se faire avec le code suivant.*

```
LOOP parcours_moyenne IN C_moyenne_matiere
  IF moyenne_etudiant.CODE = 'BD'
  THEN
        parcours_moyenne.MOYENNE:= parcours_moyenne.MOYENNE + 1 ;
  END IF ;
END LOOP ;
```
□

La gestion automatique de curseur simplifie le codage de gestion de curseur. Cependant, il faut la réserver aux cas où tous les enregistrements du curseur doivent être lus. Dans les autres cas, il faut préférer une gestion manuelle.

6.5 - Paramétrer des curseurs

Lorsque plusieurs curseurs sont déclarés avec la même requête de définition hormis des constantes de sélection, il est possible de les remplacer par un curseur plus générique et paramétré. La déclaration d'un tel curseur doit intégrer la spécification des paramètres.
La syntaxe de la déclaration devient :

```
CURSOR <nom_curseur> (<parametre1> <type1>[, <parametre2> <type2> …])
IS <requete_SQL> ;
```

À l'ouverture d'un curseur paramétré (c'est à ce moment que la requête est effectivement évaluée sur la base), la valeur du ou des paramètres doit être indiquée avec la syntaxe suivante :

```
OPEN <nom_curseur>(<valeur1>[, <valeur2> …]) ;
```

ou si une boucle **FOR** de parcours automatique est utilisée, les paramètres sont passés comme suit :

```
FOR <variable_parcours> IN <nom_curseur>(<valeur1>[, <valeur2> …]) …
```

Exemple *: supposons que le développeur travaille sur les moyennes générales par matière. Un curseur paramétré est défini comme suit.*

```
CURSOR C_moyenne_par_matiere (code_mat VARCHAR2(4))
       SELECT  NUM_ET, CODE,
       ( NVL(MOY_TEST,0) * COEFF_TEST + NVL(MOY_CC,0)* COEFF_CC )/
       ( COEFF_TEST+COEFF_CC ) MOYENNE
       FROM  NOTATION WHERE  CODE = code_mat
       ORDER  BY  MOYENNE DESC ;
```

Le curseur peut alors être ouvert en spécifiant une matière particulière.

```
OPEN C_moyenne_par_matiere ('BD') ;
```

□

Exemple *: on suppose qu'il existe, dans la relation MODULE, un attribut calculé MOYENNE_MAX qui devra donner la meilleure moyenne obtenue dans chaque matière. Le programme ci-après (un bloc anonyme) calcule les valeurs de cet attribut. Il utilise le curseur paramétré de l'exemple précédent ainsi qu'un curseur contenant le code de toutes les matières.*

```
DECLARE

CURSOR liste_matiere IS
  SELECT CODE
  FROM MODULE
  WHERE CODE NOT IN (
       SELECT CODEPERE
       FROM MODULE
       WHERE CODEPERE IS NOT NULL);

CURSOR C_moyenne_par_matiere (code_mat NOTATION.CODE%TYPE) IS
    SELECT  CODE, NUM_ET,
    (NVL(MOY_TEST,0) + NVL(MOY_CC,0))/(COEFF_CC+COEFF_TEST)  MOYENNE
    FROM  NOTATION
    WHERE  CODE = code_mat
    ORDER  BY  MOYENNE DESC  ;

moyenne_max C_moyenne_par_matiere%ROWTYPE ;
```

```
BEGIN
FOR parcours_matiere IN liste_matiere
LOOP
     /* Le curseur paramétré est ouvert en lui passant en paramètre le
     code de la matière courante */
     OPEN C_moyenne_par_matiere(parcours_matiere.CODE) ;
     FETCH C_moyenne_par_matiere INTO moyenne_max ;
     IF C_moyenne_par_matiere%FOUND
     THEN
           UPDATE MODULE SET MOYENNE_MAX = moyenne_max.MOYENNE
           WHERE CODE = parcours_matiere.CODE ;
     END IF ;
     CLOSE C_moyenne_par_matiere ;
END LOOP ;
END ;
```

Les enregistrements dans le curseur `C_moyenne_par_matiere` étant triés, il suffit de lire le premier de ces enregistrements (s'il existe) puis de mettre à jour l'attribut `MOYENNE_MAX` dans la relation MODULE. □

7 - Blocs anonymes, procédures et fonctions

Tout bloc ou programme PL/SQL est soit un bloc anonyme, soit une procédure, soit une fonction. Les blocs de ces diverses natures peuvent être imbriqués. Un bloc anonyme, tout comme une procédure ou une fonction, peut posséder ses propres procédures ou fonctions (définies dans sa partie déclarative). Son corps peut intégrer des blocs anonymes imbriqués.

7.1 - Spécification de blocs anonymes

Comme indiqué au paragraphe 1, un bloc anonyme n'a pas d'en-tête, il débute simplement par le mot clef **`DECLARE`** ou éventuellement par **`BEGIN`** s'il n'intègre aucune déclaration spécifique.
Comme il est dépourvu de nom, un bloc anonyme ne peut pas être appelé par un autre bloc.

Les blocs anonymes ont deux utilisations :
- soit ils font office de scripts PL/SQL, ils réalisent une fonctionnalité particulière et sont déclenchés par l'utilisateur ;
- soit ils permettent de mieux structurer des programmes et de mieux gérer la portée des variables et exceptions. Ils sont alors intégrés dans le corps d'un module de niveau supérieur.

Quelle que soit leur utilisation, les blocs anonymes peuvent contenir des déclarations de procédures ou fonctions locales au bloc et ils peuvent faire des appels à des procédures ou fonctions[5].

[5] Ces procédures ou fonctions appelées dans le bloc peuvent être soit locales, soit prédéfinies en PL/SQL (e.g. **`floor`**, **`sysdate`**, **`to_char`**…), soit stockées (Cf. paragraphes 7.2.2 et 7.3.2).

L'imbrication de blocs dans le corps d'un autre bloc anonyme est illustré par la figure 2. Le principe est le même s'il s'agit d'une procédure ou d'une fonction. La portée des variables et exceptions d'un bloc concerne tout le bloc y compris ses éventuels blocs imbriqués. L'utilisation d'étiquettes permet une meilleure lisibilité lors de l'imbrication de blocs anonymes.

Les composants déclarés dans un bloc imbriqué ne sont pas accessibles pour les blocs de niveau supérieur. En particulier si des exceptions sont levées et traitées dans un bloc imbriqué, l'exécution se poursuit en considérant le code qui suit le bloc imbriqué.

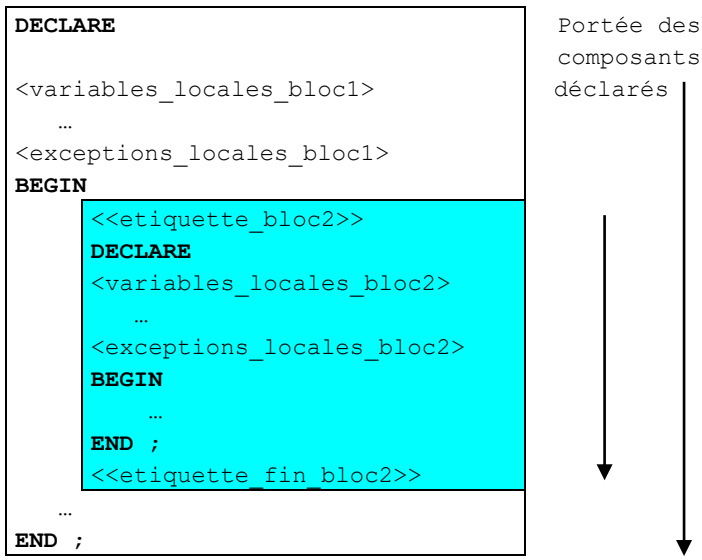

Figure 3 : Portée des variables dans des blocs imbriqués

Exemple *: on suppose qu'il existe un attribut NB_ABSENCES dans la relation NOTATION. Tout étudiant ayant plus de deux absences dans la matière de code BD verra diminuer sa note de contrôle continu. De plus tout étudiant ayant 20 en contrôle continu de BD aura une augmentation de sa note de test. Le bloc anonyme donné ci-dessous permet de réaliser ces opérations.*

```
DECLARE
    CURSOR absent_bd IS
        SELECT NUM_ET FROM NOTATION WHERE CODE ='BD' AND NB_ABSENCES > 2 ;

    CURSOR controle_cc_20 IS
        SELECT NUM_ET FROM NOTATION WHERE CODE ='BD' AND MOY_CC = 20 ;
BEGIN
    FOR etudiant_absent IN absent_bd LOOP
        UPDATE NOTATION SET MOY_CC = greatest(MOY_CC - 1, 0)
        WHERE NUM_ET = etudiant_absent.NUM_ET ;
    END LOOP ;

    FOR etudiant_tb IN controle_cc_20 LOOP
          UPDATE NOTATION SET MOY_TEST = least(MOY_TEST + 1, 20)
          WHERE NUM_ET = etudiant_tb.NUM_ET ;
    END LOOP ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN NULL ;
END ;
```

Le problème du bloc proposé est que s'il n'existe pas d'étudiant ayant plus de deux absences, l'exception ***NO_DATA_FOUND*** *est levée (lors de l'évaluation de la requête de définition du curseur), le programme s'arrête et aucun bonus n'est accordé pour une note de contrôle continu à 20. Pour remédier à ce problème, le même traitement est réalisé avec une imbrication de blocs.*

```
DECLARE
     CURSOR absent_bd IS
          SELECT NUM_ET, MOY_CC FROM NOTATION
          WHERE CODE='BD' AND NB_ABSENCES > 2 ;

     CURSOR controle_cc_20 IS
          SELECT NUM_ET, MOY_TEST FROM NOTATION
          WHERE CODE='BD' AND MOY_CC = 20 ;
BEGIN
     <<penalisation_absents>>
     BEGIN
          FOR etudiant_absent IN absent_bd LOOP
               UPDATE NOTATION SET MOY_CC = greatest(MOY_CC - 1, 0)
               WHERE NUM_ET = etudiant_absent.NUM_ET ;
          END LOOP ;
     EXCEPTION
          WHEN NO_DATA_FOUND THEN NULL ;
     END ;
     <<fin_penalisation_absents>>

     <<gratification_bon_cc>>
     BEGIN
     FOR etudiant_tb IN controle_cc_20 LOOP
          UPDATE NOTATION SET MOY_TEST = least(MOY_TEST + 1, 20)
          WHERE NUM_ET = etudiant_tb.NUM_ET ;
     END LOOP ;
     EXCEPTION
          WHEN NO_DATA_FOUND THEN NULL ;
     END ;
     <<fin_gratification_bon_cc>>
END ;
```

La portée de l'exception du premier bloc imbriqué est limitée à ce premier bloc. Si l'exception est levée, le programme continue son exécution par le deuxième bloc imbriqué. □

7.2 - Création et utilisation de procédures

Une procédure est un bloc PL/SQL qui est nommé et peut donc être appelé. Il est généralement (mais pas obligatoirement) pourvu de paramètres dont les valeurs doivent lui être passées lors d'un appel. La syntaxe de création d'une procédure est donnée ci-après.

```
PROCEDURE <nom_procedure>[(<specification_parametre>)]
IS
       [<liste_declarations>]
BEGIN
       <liste_instructions>
[EXCEPTION
       <gestion_exception> ]
END  [<nom_procedure>] ;
```

Si la procédure n'a pas besoin de type, variable, constante, curseur, exception, procédure ou fonction spécifique, `<liste_declarations>` est omise. De même, si elle ne gère aucune

exception, la partie <gestion_exception> ne figure pas dans le corps de la procédure. La partie <liste_instructions> doit contenir au moins une instruction.

<specification_parametre> est une liste définissant les divers paramètres nécessaires à la procédure. Pour chacun d'eux, on doit préciser le nom, le mode d'utilisation, le type et éventuellement la valeur par défaut, de la manière suivante.

```
<nom_parametre> <mode> <nom_type> [:= <valeur_defaut>]
```

<mode> peut prendre les valeurs **IN**, **OUT** ou **IN OUT** suivant que la procédure lit seulement le paramètre, l'écrit seulement ou le lit et l'écrit. Il est possible d'affecter une valeur par défaut à des paramètres **IN** ou **IN OUT**. Un paramètre **OUT** doit être valorisé dans le code du bloc et ne peut pas avoir de valeur par défaut. De plus, il doit toujours figurer à gauche d'une affectation. On ne peut pas l'utiliser pour calculer une autre variable.
La spécification du type des paramètres peut inclure des références avec **%TYPE** ou **%ROWTYPE**.

***Exemple** : la procédure suivante effectue le formatage des paramètres qui lui sont fournis, en mettant le premier en majuscule et le second en minuscule sauf l'initiale.*

```
PROCEDURE formater_nom_prenom
        (nom IN OUT ETUDIANT.NOM_ET%TYPE,
        prenom IN OUT ETUDIANT.PRENOM_ET%TYPE)
IS
BEGIN
        nom := UPPER(nom) ;
        prenom := INITCAP(prenom);
END formater_nom_prenom ;
```

□

Une procédure peut intégrer des blocs anonymes imbriqués pour une meilleure gestion de la portée des variables et exceptions.

L'appel d'une procédure se fait en indiquant son nom et ses paramètres. Un paramètre ayant des valeurs par défaut peut être omis. Le passage de paramètres peut se faire avec une notation positionnelle ou une notation fléchée. Dans le premier cas, les arguments de l'appel de la procédure sont donnés dans le même ordre que les paramètres dans la définition de la procédure.

```
<nom_procedure>(argument1, argument2, …) ;
```

Mais ceci peut poser problème si les arguments omis dans l'appel ne sont pas les derniers. Avec la notation fléchée, un appel de procédure prend la forme :

```
<nom_procedure>(parametre1 => argument1, parametre2 => argument2, …) ;
```

***Exemple** : considérons l'entête de procédure suivant.*

```
PROCEDURE formater(
        formatc IN VARCHAR2(10) DEFAULT 'MAJUSC',
        chaine IN OUT VARCHAR2)
```

Les appels de cette procédure, indiqués ci-dessous sont valides, en supposant que les variables utilisées sont correctement déclarées et manipulées.

```
formater('MINUSC', prenom) ;
formater(chaine => prenom, formatc =>'MINUSC') ;
formater(chaine => nom) ;
```

□

7.2.1 - Procédures locales

Une procédure utilisée par un programme particulier est intégrée dans le bloc constituant ce programme, elle est dite locale. Sa définition apparaît alors dans la partie déclarative du bloc et elle est appelée dans le corps de ce bloc.

***Exemple** : la procédure effectuant le formatage des noms et prénoms est intégrée dans un bloc anonyme comme illustré par le programme suivant.*

```
DECLARE
  CURSOR C_etudiant  IS
       SELECT  NUM_ET, NOM_ET, PRENOM_ET  FROM  ETUDIANT ;

 PROCEDURE formater_nom_prenom
       (nom IN OUT ETUDIANT.NOM_ET%TYPE,
       prenom IN OUT ETUDIANT.PRENOM_ET%TYPE)
 IS
 BEGIN
       nom := UPPER(nom) ;
       prenom := INITCAP(prenom);
 END formater_nom_prenom ;

BEGIN
  FOR un_etudiant IN C_etudiant
  LOOP
     formater_nom_prenom(un_etudiant.NOM_ET, un_etudiant.PRENOM_ET) ;
     UPDATE ETUDIANT
     SET NOM_ET = un_etudiant.NOM_ET, PRENOM_ET = un_etudiant.PRENOM_ET
     WHERE NUM_ET = un_etudiant.NUM_ET ;
  END LOOP ;
EXCEPTION
  WHEN NO_DATA_FOUND THEN NULL ;
END ;
```

□

7.2.2 - Procédures stockées

Lorsqu'une procédure est utilisée par différents programmes, il est intéressant d'en faire « un objet ORACLE » en l'intégrant dans la base de données. On parle alors de ***procédure stockée***. Après cette opération, la procédure peut être appelée directement par différents programmes PL/SQL. Le stockage d'une procédure consiste plus précisément à la créer : le code associé est stocké dans la base et compilé mais la procédure n'est pas exécutée. La commande permettant cette création est :

```
CREATE PROCEDURE <nom_procedure> <specification_procedure> ;
```

où `<specification_procedure>` est la définition de procédure donnée précédemment. Si la procédure doit être recréée pour cause d'erreurs de compilation par exemple, l'ancienne version doit être détruite au préalable par :

```
DROP PROCEDURE <nom_procedure> ;
```

En phase de mise au point, il est donc préférable d'utiliser la commande suivante qui crée une procédure ou substitue son ancienne version par une nouvelle.

```
CREATE OR REPLACE PROCEDURE <nom_procedure> <specification_procedure> ;
```

Exemple *: la procédure suivante de formatage de chaînes de caractères met la chaîne passée en paramètre soit en majuscule, soit en minuscule, soit en minuscule avec l'initiale en majuscule. Elle est stockée dans la base par :*

```
CREATE OR REPLACE PROCEDURE formater_chaine
       (chaine_traitee IN OUT VARCHAR2,
       type_carac IN VARCHAR2 DEFAULT 'MAJUSC')
IS
mauvais_format EXCEPTION ;
BEGIN
       IF chaine_traitee IS NOT NULL
       THEN
             IF UPPER(type_carac) LIKE 'MAJUSC%'
             THEN
                   chaine_traitee := UPPER(chaine_traitee) ;
             ELSIF UPPER(type_carac) LIKE 'MINUSC%'
             THEN
                   chaine_traitee := LOWER(chaine_traitee);
             ELSIF UPPER(type_carac) LIKE 'INIT%'
             THEN
                   chaine_traitee := INITCAP(chaine_traitee);
             ELSE
                   RAISE mauvais_format ;
             END IF ;
       END IF ;
EXCEPTION
       WHEN mauvais_format
       THEN afficher('le format n''est pas valide') ;
END formater_chaine ;
```

Remarquons que le deuxième argument de la procédure peut être donné indifféremment en minuscule ou majuscule (utilisation de la fonction ***UPPER*** *dans les conditions) et que l'utilisation de* ***LIKE*** *autorise plusieurs possibilités pour le deuxième argument. Les appels suivants sont tous correctement traités :*

```
formater_chaine(nom, 'majuscules');          formater_chaine(nom, 'majusc');
formater_chaine(nom, 'MAJUSCULE');           formater_chaine(nom, 'initiales');
formater_chaine(nom);                        formater_chaine(nom, 'Init');
```

Après cette création, n'importe quel programme peut utiliser la procédure.

```
DECLARE
  CURSOR C_etudiant  IS
       SELECT  NUM_ET, NOM_ET, PRENOM_ET  FROM  ETUDIANT ;

BEGIN
  FOR un_etudiant IN C_etudiant
  LOOP
    formater_chaine(un_etudiant.NOM_ET, 'majuscules') ;
    formater_chaine(un_etudiant.PRENOM_ET, 'initiale') ;

    UPDATE ETUDIANT
    SET NOM_ET = un_etudiant.NOM_ET, PRENOM_ET = un_etudiant.PRENOM_ET
    WHERE NUM_ET = un_etudiant.NUM_ET ;
  END LOOP ;
EXCEPTION
  WHEN NO_DATA_FOUND THEN NULL ;
END ;
```

□

7.3 - Création et utilisation de fonctions

La syntaxe de création d'une fonction est proche de celle d'une procédure mais elle inclut une clause **RETURN** dans la spécification de son en-tête et un ou plusieurs ordres **RETURN** parmi ses instructions.

```
FUNCTION <nom_fonction>[(<specification_parametre>)]
RETURN <type_donnee>
IS
       [<liste_declarations>]
BEGIN
       <liste_instructions>
[EXCEPTION
       <gestion_exception> ]
END  [<nom_fonction>] ;
```

Si la fonction n'a besoin d'aucune déclaration locale, `<liste_declarations>` est omise. De même, si elle ne gère aucune exception, la partie `<gestion_exception>` ne figure pas dans le corps de la fonction. La partie `<liste_instructions>` doit contenir au moins une instruction, ne serait-ce que l'ordre **RETURN** restituant le résultat.

`<specification_parametre>` est une liste définissant les divers paramètres nécessaires à la fonction de la même manière que pour une procédure. Bien sûr il ne faut pas utiliser de paramètre de mode **OUT** ou **IN OUT** puisque le résultat de la fonction est la variable restituée par l'ordre **RETURN**.
`<type_donnee>` est un des types scalaires ou composés valides en PL/SQL. Il peut être un type référencé mais ne peut être ni un curseur ni une exception.

***Exemple** : la fonction suivante rend la meilleure note de test obtenu dans une matière passée en paramètre.*

```
FUNCTION meilleure_note(code_matiere IN MODULE.CODE%TYPE)
RETURN NOTATION.NOTE_TEST%TYPE
IS
     note_max NOTATION.MOY_TEST%TYPE ;
BEGIN
     SELECT MAX(MOY_TEST) INTO note_max
     FROM NOTATION
     WHERE CODE = code_matiere ;
     RETURN(note_max) ;
END meilleure_note ;
```

□

Le corps d'une fonction peut contenir plusieurs ordres **RETURN** mais un seul sera exécuté puisque dès l'exécution d'un **RETURN**, la fonction se termine et rend la main au bloc PL/SQL appelant.

L'appel d'une fonction peut permettre l'affectation d'une valeur à une variable, l'attribution d'une valeur par défaut à une variable ou l'évaluation d'une condition.

```
<nom_variable> <nom_type> DEFAULT <nom_fonction>[(<liste_parametres>)];
<nom_variable> := <nom_fonction>[(<liste_parametres>)] ;
IF <nom_variable> <comparateur> <nom_fonction>[(<liste_parametres>)] THEN … ;
```

L'appel d'une fonction peut aussi permettre la mise à jour d'attributs de la base ou la sélection de

tuples dans un **UPDATE** ou **DELETE**.

7.3.1 - Fonctions locales

Comme pour les procédures, les fonctions peuvent être intégrées dans des blocs PL/SQL. Des fonctions locales doivent être utilisées chaque fois qu'elles réalisent une opération spécifique à un programme.

***Exemple** : en utilisant la fonction définie dans l'exemple précédent, le bloc donné ci-après met à jour l'attribut calculé NOTE_TEST_MAX que l'on a préalablement ajouté à la relation MODULE.*

```
DECLARE
CURSOR liste_matiere IS
   SELECT CODE FROM MODULE WHERE CODE NOT IN
        (SELECT CODEPERE FROM MODULE);

FUNCTION meilleure_note(code_matiere IN MODULE.CODE%TYPE)
RETURN NOTATION.MOY_TEST%TYPE
IS
     note_max NOTATION.MOY_TEST%TYPE ;
BEGIN
     SELECT MAX(MOY_TEST) INTO note_max FROM NOTATION
     WHERE CODE = code_matiere ;
     RETURN(code_matiere) ;
END meilleure_note ;

BEGIN
FOR parcours_matiere IN liste_matiere
     UPDATE MODULE
     SET NOTE_TEST_MAX = meilleure_note(parcours_matiere.CODE)
     WHERE CODE = parcours_matiere.CODE ;
END LOOP ;
END ;
```

□

7.3.2 - Fonctions stockées

Exactement comme les procédures, les fonctions peuvent être créées comme des objets ORACLE et elles deviennent des fonctions stockées. La syntaxe de création d'une telle fonction est donnée ci-dessous.

```
CREATE [OR REPLACE] FUNCTION <nom_fonction> <specification_fonction> ;
```

Lorsqu'un traitement est réalisé dans plusieurs programmes PL/SQL, il est intéressant d'en faire une fonction stockée disponible pour tous.
Un autre intérêt des fonctions stockées est qu'elles peuvent être utilisées, comme n'importe quelle fonction prédéfinie non seulement dans un bloc PL/SQL mais aussi dans toute requête SQL.

***Exemple** : la fonction `meilleure_note` a été créée dans ORACLE. La requête suivante peut être formulée.*

```
SELECT * FROM NOTATION WHERE MOY_TEST = meilleure_note('BD');
```

□

7.4 - Surcharge des procédures et fonctions

Les procédures et fonctions peuvent être surchargées mais uniquement dans la partie déclarative

d'un bloc ou dans un package (Cf. paragraphe 8). Le choix, parmi les blocs surchargés, de celui qui doit être exécuté est guidé par les paramètres. Au moins un de ces paramètres doit différer par la famille de son type entre les programmes surchargés, par exemple être de types alphabétique et numérique. Les types **CHAR** et **VARCHAR2** sont considérés comme de la même famille et la surcharge ne peut pas s'appliquer.

***Exemple** : les deux procédures surchargées dont l'en-tête est donné ci-après sont valides.*

```
PROCEDURE formater
  (chaine_traitee IN OUT VARCHAR2,
  type_carac IN VARCHAR2 DEFAULT 'MAJUSC')
IS
  …

PROCEDURE formater(nombre IN OUT NUMBER)
IS
…
```
□

7.5 - Affichage de messages

Les programmes PL/SQL peuvent retourner des messages à l'utilisateur. Cette opération d'écriture est plus précisément réalisée par une procédure **PUT_LINE** d'un package prédéfini de PL/SQL : **DBMS_OUTPUT**. L'argument donné à la procédure est le texte du message à afficher. L'appel de cette procédure dans le code d'un bloc stocké se fait donc par :

```
DBMS_OUTPUT.PUT_LINE('texte du message') ;
```

***Exemple** : la gestion d'exception, dont le code suit, permet d'envoyer un message d'explication à l'utilisateur de la procédure.*

```
CREATE OR REPLACE PROCEDURE classement IS
BEGIN
      …
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN DBMS_OUTPUT.PUT_LINE('Pas d''etudiant extrait de la base');
END classement ;
```
□

Pour que les messages soient affichés, il faut utiliser, sous SQL*PLUS, la commande ORACLE suivante : **set serveroutput on**

8 - Les packages

Les packages sont utilisés pour regrouper dans une même unité de traitements tous les blocs PL/SQL développés pour répondre à un même objectif global ou implémenter une certaine fonctionnalité. ORACLE utilise des packages pour implémenter les fonctionnalités prédéfinies (par exemple, opérateurs et fonctions PL/SQL sont définis dans le package appelé STANDARD et la procédure d'affichage PUT_LINE dans le package DBMS_OUTPUT). Évidemment les packages sont un aspect clef de la programmation modulaire. Tout package comporte une partie spécification et un corps, compilés séparément.
La partie spécification consiste en la déclaration de tous les éléments publics (types, variables,

curseurs, procédures, fonctions) qui peuvent être référencés et donc utilisés à l'extérieur du package. Cette partie doit donner toutes les informations nécessaires à un développeur pour utiliser le package sans connaître les détails d'implémentation. La syntaxe à respecter est la suivante.

```
PACKAGE <nom_package>
IS
      [<liste_declarations>]
      [<specification_curseurs>]
      [<specification_procedures_fonctions>]
END [<nom_package>] ;
```

<liste_declarations> est une liste de déclarations de types, variables et/ou constantes telles qu'elles ont été présentées au paragraphe 2.
<specification_procedures_fonctions> est la liste de tous les entêtes des procédures et fonctions publiques respectant la syntaxe décrite au paragraphe 7.
<specification_curseurs> est la liste de spécifications de curseurs. Chaque spécification doit indiquer les paramètres éventuels du curseur et la structure de l'enregistrement retourné par un **FETCH**, d'où la présence obligatoire d'une clause **RETURN**. Chaque déclaration de curseur prend la forme :

```
CURSOR <nom_curseur>[(liste_parametres)]
RETURN <nom_type_enreg> ;
```

où <nom_type_enreg> peut être la référence à la structure d'une table sous la forme <nom_relation>**%ROWTYPE** ou un type d'enregistrement préalablement défini.
Pour référencer les éléments publics d'un package, en dehors de celui-ci, il suffit d'indiquer leur nom préfixé par le nom du package (e.g. **DBMS_OUTPUT.PUT_LINE**). La commande SQL de création de synonyme peut être utilisée pour s'épargner la frappe de noms très longs.

***Exemple** : le package gestion_notes est conçu pour calculer les moyennes générales des étudiants, effectuer le classement et éditer les bulletins de notes.*

```
PACKAGE gestion_notes
IS
     note_max NOTATION.MOY_TEST%TYPE ;

     TYPE T_moyenne_etudiant IS RECORD
        (numero ETUDIANT.NUM_ET%TYPE,
        moyenne NOTATION.MOY_TEST%TYPE);

     CURSOR C_moyenne_generale
     RETURN T_moyenne_etudiant;

     CURSOR C_moyenne_par_matiere (code_mat NOTATION.CODE%TYPE)
     RETURN NOTATION%ROWTYPE ;

     FUNCTION meilleure_note(code_matiere IN NOTATION.CODE%TYPE)
     RETURN NOTATION.MOY_TEST%TYPE ;

     FUNCTION moins_bonne_note(code_matiere IN NOTATION.CODE%TYPE)
     RETURN NOTATION.MOY_TEST%TYPE ;

     PROCEDURE classement(ANNEE IN ETUDIANT.ANNEE%TYPE);

     PROCEDURE edition_bulletins(ANNEE IN ETUDIANT.ANNEE%TYPE);
```

```
END gestion_notes ;
```
□

Le corps d'un package contient la déclaration locale de tous les éléments privés, le code associé aux procédures et fonctions qu'elles soient publiques ou privées, la définition de curseurs et enfin éventuellement une liste d'instructions exécutées une fois pour initialiser le package. Un package peut donc ne pas avoir de corps si sa partie spécification ne contient que des types ou variables publics. La syntaxe utilisée est la suivante :

```
PACKAGE BODY <nom_package>
IS
      [<liste_declarations_privees>]
      [<definition_curseurs>]
      [<definition_procedures_fonctions>]
BEGIN
      [<liste_instructions_initialisation>] ;
[EXCEPTION
     <gestion_exceptions>] ;

END [<nom_package>] ;
```

***Exemple** : le corps du package `gestion_notes` est partiellement donné ci-après.*

```
PACKAGE BODY gestion_notes
IS

     CURSOR C_moyenne_generale
        SELECT  NUM_ET,
        AVG((NVL(MOY_TEST,0)+NVL(MOY_CC,0))/(COEFF_CC+COEFF_TEST) MOY_G
        FROM  NOTATION
        GROUP BY  NUM_ET
        ORDER  BY  MOY_G   ;

     CURSOR C_moyenne_par_matiere (code_mat VARCHAR2(4))
        SELECT  NUM_ET, CODE,
        (NVL(MOY_TEST,0)+ NVL(MOY_CC,0))/(COEFF_CC+COEFF_TEST) MOYENNE
        FROM  NOTATION
        WHERE  CODE = code_mat
        ORDER  BY  MOYENNE   ;

     FUNCTION meilleure_note(code_matiere IN NOTATION.CODE%TYPE)
     RETURN NOTATION.MOY_TEST%TYPE
     IS
     BEGIN ;
        …
     END meilleure_note ;

     FUNCTION moins_bonne_note(code_matiere IN NOTATION.CODE%TYPE)
     RETURN NOTATION.MOY_TEST%TYPE
     IS
     BEGIN ;
        …
     END moins_bonne_note ;

     PROCEDURE classement(ANNEE IN ETUDIANT.ANNEE%TYPE)
     IS
     BEGIN ;
        …
     END classement ;
     …
```

```
END gestion_notes ;
```

La création de package se fait par :

```
CREATE [OR REPLACE] PACKAGE <nom_package> <specification_package> ;
CREATE {OR REPLACE] PACKAGE BODY <nom_package> <package_body>;
```

9 - Compilation et suppression de blocs stockés

Pour détruire un bloc stocké, les ordres sont les suivants.

```
DROP PROCEDURE <nom_procedure> ;
DROP FUNCTION <nom_fonction> ;
DROP PACKAGE <nom_package> ;
DROP PACKAGE BODY <nom_package>;
```

Si pour des raisons de modifications structurelles de la base de données, des programmes stockés doivent être recompilés, il faut utiliser les ordres suivants (pour plus de détails voir le paragraphe 11.2).

```
ALTER PROCEDURE <nom_procedure> COMPILE ;
ALTER FUNCTION <nom_fonction> COMPILE ;
ALTER PACKAGE <nom_package> COMPILE ;
```

10 - SQL dynamique natif

Les programmes PL/SQL vus jusqu'à présent effectuent une tâche spécifique et prédictible, qui ne varie pas d'une exécution à l'autre. Ils intègrent des ordres SQL sur la base de données qui sont « prédéfinis » par le développeur. Pour ces raisons, ils sont qualifiés de statiques. Mais il est possible qu'un programme construise et exécute « à la volée » des ordres SQL. On parle alors de SQL dynamique[6] et de programmes dynamiques. Par exemple, un programme peut supprimer des tuples dans une relation sans que le développeur du programme connaisse a priori la relation concernée et la condition de sélection des tuples à détruire. Ces éléments seront fournis au programme, par passage de paramètres, seulement au moment de l'exécution. Dans de tels cas, les ordres SQL à exécuter sont construits par le programme.

Les programmes statiques ont des avantages par rapport aux programmes dynamiques. Généralement leurs performances sont meilleures et leur compilation préalable permet de vérifier que les éléments du schéma de la base sont corrects ainsi que les privilèges d'accès. Mais les programmes statiques ont des limites qui peuvent être dépassées en utilisant SQL dynamique.
Les programmes dynamiques permettent, entre autres, d'utiliser l'aspect définition de données (LDD) de SQL, d'exécuter des requêtes dont les clauses varient ou encore de choisir quel bloc PL/SQL appeler au moment de l'exécution.

10.1 - Mises à jour et ordres SELECT simples

[6] Avec la version 8i d'ORACLE, le SQL dynamique est natif : les ordres dynamiques peuvent être directement insérés dans le code PL/SQL. Les versions antérieures nécessitent l'utilisation du package DBMS_SQL [1].

Les ordres **SELECT** simples sont ceux ne retournant qu'au plus un résultat (valeur ou tuple). De telles requêtes ainsi que les opérations de mise à jour en SQL dynamique (sans le ; final) ou les blocs PL/SQL (avec le ; final) sont stockés dans une chaîne de caractères construite par le programme et la commande **EXECUTE IMMEDIATE** est utilisée pour analyser et exécuter ces ordres dynamiques.

***Exemple** : la procédure suivante admet en entrée deux paramètres : le nom d'un type d'objet ORACLE (relation, vue, index…) et le nom d'un objet de ce type. Son rôle est de détruire l'objet spécifié. Il faut donc, en fonction du type de l'objet qui lui est passé, qu'elle réalise des ordres **DROP TABLE**, **DROP VIEW**, **DROP INDEX**… (rappel : || est l'opérateur de concaténation de chaînes).*

```
CREATE PROCEDURE detruire(nom_type IN VARCHAR2, nom_objet IN VARCHAR2)
IS
     type_obj USER_OBJECTS.OBJET_TYPE%TYPE ;
BEGIN
     SELECT OBJECT_TYPE INTO type_obj FROM USER_OBJECTS
     WHERE OBJECT_TYPE = UPPER(nom_type) AND OBJECT_NAME = UPPER(nom_objet);

     EXECUTE IMMEDIATE 'DROP ' || nom_type || ' ' || nom_objet ;

EXCEPTION
     WHEN NO_DATA_FOUND
         THEN DBMS_OUTPUT.PUT_LINE('Il n''existe pas d''objet de ce type');
END detruire ;
```

La procédure `detruire` peut être appelée des manières suivantes :

```
detruire('TABLE', 'notation');
detruire('view', 'consult_prof');
detruire('PROCEDURE', 'formater_chaine');
```

□

Les programmes dynamiques construisent des chaînes qui sont ensuite exécutées comme des ordres SQL ou PL/SQL. Dans ces chaînes, les séquences de caractères correspondant aux éléments d'une base de données doivent respecter les mêmes contraintes pour ne pas produire d'erreur. Par exemple les identificateurs des composants du schémas ont une longueur maximale, doivent commencer par une lettre et peuvent être donnés indifféremment en minuscule ou majuscule (Cf. exemple précédent). En revanche, pour les valeurs stockées, la distinction minuscule/majuscule opère. Il faut donc que les programmes gèrent les divers cas de figure. Enfin, si une valeur de type alphanumérique doit être insérée dans la chaîne représentant l'ordre dynamique, elle doit l'être entre '' et '' (Cf. exemple suivant).

***Exemple** : la procédure suivante permet à l'utilisateur de supprimer, dans une relation, des tuples répondant à une condition. Le nom de la relation et la condition lui sont passées en paramètres.*

```
CREATE PROCEDURE supprimer_tuples
     (nom_relation IN VARCHAR2, condition IN VARCHAR2 DEFAULT NULL)
IS
ordre_dyn VARCHAR2(100);
BEGIN
     /* si une condition est passée en paramètre, la construction de l'ordre
     dynamique débute par la mise en forme de la clause WHERE */
```

```
     IF condition IS NOT NULL
     THEN
          ordre_dyn := ' where '|| condition ;
     END IF ;

     /* L'ordre dynamique est calculé par une concaténation de 3 chaînes. Si
     le dernier argument est NULL (pas de clause WHERE), le résultat est la
     concaténation des deux premiers */

     ordre_dyn:='delete from '|| nom_relation || ordre_dyn ;
     EXECUTE IMMEDIATE ordre_dyn ;

END supprimer_tuples ;
```

Les appels suivants de la procédure `supprimer_tuples` *sont valides.*

```
     supprimer_tuples('notation');
     supprimer_tuples('etudiant', 'num_et = 2304');
     supprimer_tuples('prof', 'nom_prof = ''DUBOIS'' ');
     supprimer_tuples('notation', 'num_et = 2422 and code = ''BD'' ');
```

□

Exemple *: la procédure suivante admet en entrée deux paramètres : le nom d'une relation et une liste d'attributs (séparés par des virgules). Son rôle est de créer une nouvelle relation agrégeant les données (en fait réalisant un comptage) de la table passée en paramètre selon les attributs indiqués.*

```
CREATE OR REPLACE PROCEDURE creer_relation_agr
     (nom_relation IN VARCHAR2, liste_attributs IN VARCHAR2 DEFAULT NULL)
IS
     ordre_dyn VARCHAR2(200);
BEGIN
     ordre_dyn:='create table   '|| nom_relation || '_agr ' || 'as select
     '||liste_attributs ||', count(*) cptage  from '||nom_relation ||' group
     by '||liste_attributs ;
     DBMS_OUTPUT.PUT_LINE(ordre_dyn) ;
     EXECUTE IMMEDIATE ordre_dyn ;

END creer_relation_agr ;
```

Supposons que la procédure soit appelée avec les paramètres suivants :

```
creer_relation_agr('ENSEIGNT', 'NUM_PROF, CODE')
```

La chaîne construite dynamiquement par le programme est :

```
'create table ENSEIGNT_agr as select NUM_PROF, CODE, count(*) cptage from
ENSEIGNT group by NUM_PROF, CODE'
```

□

10.1.1 - Utilisation de EXECUTE IMMEDIATE

Comme illustré dans le premier exemple du paragraphe précédent, l'ordre **EXECUTE IMMEDIATE** permet d'effectuer des modifications de schéma dynamiquement. Il peut s'appliquer également à des mises à jour de données et des requêtes simples, i.e. retournant au plus un résultat. Sa syntaxe générale est la suivante.

```
     EXECUTE IMMEDIATE <ordre_dynamique>
     [INTO <variable1 [, variable2 …] | variable_enreg>
```

```
[USING <mode> <argument_lie> [,<mode> <argument_lie> …] ]
[RETURNING INTO <argument_lie> [,<argument_lie> …] ] ;
```

où :

- <ordre_dynamique> est une chaîne stockant un ordre SQL ou un appel de bloc PL/SQL. <ordre_dynamique> peut contenir des variables préfixées par le caractère « : ». Il ne faut pas déclarer ces variables.
- La clause **INTO** n'est utilisée que pour les requêtes **SELECT** rendant ***au plus un tuple***. Elle spécifie une ou plusieurs variables réceptrices pour ce tuple. Plus précisément, on peut indiquer une liste de variables scalaires respectant l'ordre des attributs dans le **SELECT** ou une variable enregistrement dont le type est soit défini par le développeur (**RECORD**) soit référencé via **%ROWTYPE**.
- La clause **USING** permet, au moment de l'exécution, de spécifier des arguments liés aux variables éventuellement données dans <ordre_dynamique> avec leur mode d'utilisation (**IN**, **IN OUT**, **OUT**). Par défaut, ce mode est **IN**. Ces arguments doivent être indiqués dans l'ordre d'apparition des variables associées.
- La clause **RETURNING INTO** indique des arguments liés en sortie. Dans ce cas, les ordres **INSERT**, **UPDATE** et **DELETE** stockés dans <ordre_dynamique> doivent inclure une clause :
  RETURNING <nom_attribut1[, nom_attribut2…]> **INTO** :variable1[,:variable2…]
  Les arguments sont indiqués en respectant l'ordre d'apparition des variables en sortie.
  Si cette clause est utilisée, alors la clause **USING** ne doit contenir que des arguments de mode **IN**.

SQL dynamique accepte seulement des variables et arguments dont les types sont valides en SQL. Ainsi ils ne peuvent pas être des booléens ou des tableaux (types spécifiques de PL/SQL). La seule exception est la variable enregistrement indiquée dans la clause **INTO**.
Une autre restriction concerne les objets du schéma : variables et arguments liés ne peuvent pas être utilisés pour passer le nom de tels composants à l'ordre dynamique. Il faut utiliser les paramètres des procédures et fonctions.

***Exemple** : supposons que les variables suivantes soient déclarées.*

```
ordre_dyn VARCHAR2(200) ;
un_etudiant ETUDIANT%ROWTYPE ;
numero ETUDIANT.NUM_ET%TYPE ;
prenom ETUDIANT.PRENOM_ET%TYPE ;
grp ETUDIANT.GROUPE%TYPE ;
```

Les instructions dynamiques suivantes sont valides.

```
ordre_dyn := 'select * from etudiant where num_et = 2422';
EXECUTE IMMEDIATE ordre_dyn ;

/* la requête suivante retourne dans la variable un_etudiant le tuple
sélectionné. À l'exécution, la variable numero sera substituée à :num */

ordre_dyn := 'select * from etudiant where num_et = :num';
EXECUTE IMMEDIATE ordre_dyn INTO un_etudiant USING numero;

/* la mise à jour suivante utilisant la variable grp s'applique sur
l'étudiant sélectionné grâce à numero. grp et numero sont substitués
respectivement à :gp et :num */

ordre_dyn := 'update etudiant set groupe = :gp where num_et = :num' ;
EXECUTE IMMEDIATE ordre_dyn USING grp, numero;
```

```
/* la mise à jour ci-après s'applique sur l'étudiant sélectionné, son
prénom est retourné dans la variable prenom */

ordre_dyn := 'update etudiant set groupe = :gp where num_et = :num
returning prenom_et into :prn';
EXECUTE IMMEDIATE ordre_dyn USING groupe, numero RETURNING INTO prenom; □
```

10.1.2 - Curseurs implicites

Pour traiter les requêtes SQL dynamiques vues précédemment, ORACLE utilise un curseur dit implicite appelé `SQL`. Ce curseur est doté des mêmes propriétés que les autres i.e. **`%FOUND`**, **`%NOTFOUND`**, **`%ISOPEN`** et **`%ROWCOUNT`** (Cf. paragraphe 6.3). Ces propriétés peuvent être manipulées par le programme, par exemple pour être testées après l'exécution d'un ordre **`EXECUTE IMMEDIATE`**.

Exemple *: reprenons la procédure* `supprimer_tuples`*. Les propriétés du curseur* `SQL` *sont évaluées pour l'affichage de messages différents.*

```
CREATE OR REPLACE PROCEDURE supprimer_tuples
     (nom_relation IN VARCHAR2, condition IN VARCHAR2 DEFAULT NULL)
IS
     ordre_dyn VARCHAR2(100);
BEGIN

     IF condition IS NOT NULL
     THEN
          ordre_dyn := ' where '|| condition ;
     END IF ;
     ordre_dyn:='delete from '|| nom_relation || ordre_dyn ;
     EXECUTE IMMEDIATE ordre_dyn ;
     IF SQL%FOUND THEN
          DBMS_OUTPUT.PUT_LINE
          (TO_CHAR(SQL%ROWCOUNT)|| ' tuples de ' || UPPER(nom_relation)|| '
          ont été détruits') ;
     ELSE
          DBMS_OUTPUT.PUT_LINE('Aucun tuple détruit dans '
           ||UPPER(nom_relation));
     END IF ;
END supprimer_tuples ;                                                    □
```

10.1.3 - Appel dynamique de blocs PL/SQL

De la même manière qu'avec les ordres SQL, des appels de blocs PL/SQL peuvent faire partie d'un ordre dynamique et **`EXECUTE IMMEDIATE`** est utilisé.

Exemple *: supposons qu'il existe deux versions différentes d'un procédure de calcul :* `calculer_a` *et* `calculer_b` *. Le code suivant construit l'appel à l'une des deux procédures en utilisant la version et la liste de paramètres données en arguments.*

```
CREATE OR REPLACE PROCEDURE calcul
     (version IN VARCHAR2, liste_parametre IN VARCHAR2)
IS
BEGIN
```

```
    EXECUTE IMMEDIATE 'begin calculer_' ||version||'(:lp) ; end ;'
    USING liste_parametre ;
END calcul ;
```

*Grâce à **USING**, la liste de paramètres passée en argument à la procédure est substituée à `:lp` dans l'ordre PL/SQL à exécuter dynamiquement. Avec l'appel `calcul('a','numero')` la chaîne créée par la procédure est :*
`'begin calculer_a(numero) ; end ;'` □

10.2 - Ordres SELECT rendant un ensemble de résultats

Lorsqu'une requête SQL dynamique rend un ensemble de résultats, il faut utiliser des curseurs. Comme les curseurs classiques, ces curseurs dynamiques sont pourvus des propriétés **%FOUND**, **%NOTFOUND**, **%ISOPEN** et **%ROWCOUNT** (Cf. paragraphe 6.3).
Les instructions qui permettent leur gestion répondent aux mêmes objectifs que celles utilisées pour les curseurs classiques. Ces ordres sont les suivants :

- **OPEN-FOR** qui associe un curseur à la requête dynamique, l'exécute et initialise **%ROWCOUNT** à 0. Sa syntaxe est :
  ```
  OPEN <nom_curseur> FOR <ordre_dynamique>
  [USING <argument_lie> [, <argument_lie> …] ] ;
  ```
  Des arguments liés à des variables de l'ordre SQL dynamique peuvent être spécifiés. Ils sont associés de manière positionnelle (i.e. selon l'ordre d'apparition) aux variables incluses dans `<ordre_dynamique>`.

- **FETCH** qui retourne un enregistrement du curseur. Sa syntaxe est :
  ```
  FETCH <nom_curseur> INTO <variable_receptrice> ;
  ```

- **CLOSE** qui désactive un curseur et libère la mémoire occupée. Sa syntaxe est :
  ```
  CLOSE <nom_curseur> ;
  ```

La requête SQL de définition du curseur n'étant connue qu'à l'exécution, il faut définir un type de curseur par référence au type générique **REF CURSOR** ainsi qu'une variable curseur de ce type. Ces déclarations se font de la manière suivante :
```
TYPE <type_curseur> IS REF CURSOR ;
<nom_curseur> <type_curseur> ;
```

***Exemple** : la procédure dont le code est donné ci-après permet à l'utilisateur de faire différentes requêtes de sélection sur la relation ETUDIANT en lui donnant simplement en paramètre la condition de sélection sous forme de chaîne. La requête de sélection est construite et exécutée. Son résultat est stocké dans un curseur qui est parcouru pour afficher à l'utilisateur le numéro et le nom des étudiants sélectionnés.*

```
CREATE OR REPLACE PROCEDURE requete(condition IN VARCHAR2 DEFAULT NULL)
IS
   TYPE les_etudiants IS REF CURSOR ;
   liste_etudiants les_etudiants ;
   un_etudiant ETUDIANT%ROWTYPE ;
   ordre_dyn VARCHAR2(100);


BEGIN
```

```
    IF condition IS NOT NULL
        THEN ordre_dyn := ' where '|| condition ;
    END IF ;
    ordre_dyn:= 'select * from etudiant' || ordre_dyn ;
    OPEN liste_etudiants FOR ordre_dyn ;
    LOOP
        FETCH liste_etudiants INTO un_etudiant;
        EXIT WHEN liste_etudiants%NOTFOUND ;
        DBMS_OUTPUT.PUT_LINE
        (TO_CHAR(un_etudiant.NUM_ET)|| ' '|| un_etudiant.NOM_ET);
    END LOOP;
    CLOSE liste_etudiants ;
END;
```

Les appels suivants de cette procédure rendent à l'utilisateur le résultat escompté.

```
requete('groupe = 2') ;
requete('annee = 2 and groupe = 5 ') ;
requete('groupe = 2 and adr_et = ''MARSEILLE'' ');
requete('groupe = 2 and nom_et like ''A%'' ');
```

□

11 - Conseils pratiques

Ce paragraphe explique, dans un premier temps, comment pratiquement créer des programmes PL/SQL, puis comment les mettre au point.

11.1 - Écriture de programmes PL/SQL

Concrètement un programme PL/SQL peut être directement écrit sous l'éditeur ORACLE mais, sauf cas particulier de test de procédures et fonctions stockées, cette démarche est fortement déconseillée.
Avec l'éditeur de votre choix, il vaut mieux créer un fichier contenant votre programme. Même si ce dernier est un bloc stocké, compilé et mis au point, il faut conserver le fichier de code initial pour faciliter les mises à jour ultérieures. Si le code source doit être modifié, effectuez les changements dans le fichier original et recréez le bloc stocké.
Il est intéressant d'utiliser deux extensions de fichier différentes pour distinguer les fichiers de code :

- les désignations de forme `<nom_fichier>.sql` seront utilisées pour les scripts SQL et PL/SQL,
- les désignations `<nom_fichier>.osp` seront réservées aux blocs stockés (`osp` pour Oracle Stored Procedure).

Terminez votre fichier de code par une dernière ligne contenant `/`, cela permettra une exécution immédiate au chargement du fichier sous ORACLE. Si ce n'est pas le cas, vous vous retrouvez en mode insertion dans l'éditeur d'ORACLE. Pour en sortir et exécuter le fichier chargé, tapez `/`.

Pour compiler et/ou exécuter le programme contenu dans un fichier, il suffit de le charger sous ORACLE dans l'environnement SQL*PLUS par `@nom_fichier` s'il a une extension `.sql` (exactement comme pour une requête SQL) ou `@nom_fichier.ops` sinon. Suivant la nature du programme, cette commande a des effets différents :

- Si le fichier est un script PL/SQL, le code est compilé. S'il y a des erreurs, des messages sont affichés à l'écran. En l'absence d'erreurs de compilation, le script est exécuté.
- Si le fichier contient un ordre de création de bloc stocké, le code est compilé et le bloc est

créé dans la base ce qui signifie diverses insertions automatiques dans des tables systèmes (Cf. tableau 2). Un message signale que le bloc stocké est créé éventuellement avec des erreurs de compilation. En aucun cas, le code n'est exécuté. Il le sera lors d'un appel de la procédure ou fonction. La commande ORACLE **show errors** peut être utilisée pour consulter les erreurs.

11.2 - Mise au point de programmes PL/SQL

Déboguer des programmes PL/SQL n'est pas facile car les messages d'erreur ne sont pas forcément explicites et les numéros de ligne associés aux erreurs ne correspondent pas forcément aux numéros de ligne dans le fichier de code original. Testez vos programme partie par partie et utilisez des traces. L'affichage de message (avec `DBMS_OUTPUT.PUT_LINE('texte du message')`) doit être exploité par le développeur pour tracer son programme.

Pour les blocs stockés, il existe différentes facilités.
Puisque les blocs stockés sont des objets ORACLE, ils sont décrits dans le dictionnaire, i.e. dans différentes tables (ou vues) système (Cf. tableau 2).
Il existe une vue particulière USER_ERRORS qui répertorie les erreurs de compilation des blocs dont l'utilisateur est propriétaire. Pour consulter ces erreurs, il suffit d'effectuer une requête sur la vue USER_ERRORS ou plus simplement d'utiliser **show errors**.

La vue `USER_OBJECTS` permet de consulter les informations décrivant tous les « objets » de l'utilisateur, parmi lesquels les blocs stockés qu'il a créés. Pour ces derniers, le type de l'objet prend ses valeurs dans l'ensemble : {FUNCTION, PROCEDURE, PACKAGE, PACKAGE BODY}. Pour chaque bloc créé, l'attribut OBJECT_STATUS peut prendre deux valeurs : VALID ou INVALID. En fait, cet attribut indique au système si le bloc considéré doit ou pas être recompilé avant son exécution. Pour un bloc donné, ORACLE met cet attribut à INVALID dans les deux cas suivants :

- le bloc utilise ou simplement référence dans une de ses déclarations (via **%TYPE** ou **%ROWTYPE**) un attribut d'une relation ou la relation elle-même. Or, depuis la dernière compilation du bloc, la structure de la relation a été modifiée par un **ALTER TABLE**, la table a été détruite ou renommée ;
- le bloc fait appel à des procédures ou des fonctions stockées qui ont été modifiées depuis sa dernière compilation.

Si le statut du bloc stocké est VALID et qu'un utilisateur ou un programme l'appelle, il est directement exécuté. Si le bloc est dans un statut INVALID, ORACLE le recompile automatiquement avant l'exécution. Ceci peut être gênant en terme de temps de réponse pour l'utilisateur final. Pour éviter cette re-compilation à l'exécution, le développeur peut la déclencher lui-même par les commandes :

```
ALTER PROCEDURE <nom_procedure> COMPILE ;
ALTER FUNCTION <nom_fonction> COMPILE ;
```

Pour pouvoir assurer la re-compilation automatique à l'exécution, ORACLE tient à jour d'une part les liens existant entre blocs stockés et relations du schéma et d'autre part les liens entre blocs. Ces informations sont répertoriées dans la vue USER_DEPENDENCIES dans laquelle chaque tuple décrit un lien entre un bloc stocké identifié par son nom (attribut NAME) et soit un autre bloc soit une relation ou vue (attribut REFERENCED_NAME). L'attribut REFERENCED_TYPE permet de connaître la nature de l'objet lié. Ainsi pour connaître les relations référencées ou utilisées par un bloc, il suffit de formuler une requête SQL sur la relation USER_DEPENDENCIES. De la même manière les liens entre blocs stockés peuvent être retrouvés en consultant cette vue.

Enfin, une dernière vue système peut être très utile au développeur PL/SQL. Cette vue est

USER_SOURCE qui permet de consulter le code des blocs stockés. Chaque ligne de code y correspond à un tuple. Plus précisément, un tuple est décrit par le nom du bloc auquel la ligne de code appartient, le type du bloc, le numéro de la ligne dans le bloc et le contenu de cette ligne.
Pour retrouver la ligne dans laquelle une erreur de compilation est signalée, cette vue peut être consultée de la manière suivante :

```
SELECT TEXT
FROM   USER_SOURCE
WHERE  LINE = &1 AND NAME ='&2' ;
```

***Exemple** : quelles sont les relations référencées par la fonction* `meilleure_note`.

```
SELECT REFERENCED_NAME
FROM USER_DEPENDENCIES
WHERE NAME LIKE 'MEIL%' AND REFERENCED_TYPE = 'TABLE' ;
```

Donnez pour chaque bloc invalide, la liste de ses liens avec d'autres blocs.

```
SELECT NAME, REFERENCED_NAME
FROM USER_OBJECTS OBJ1, USER_OBJECTS OBJ2, USER_DEPENDENCIES DEP
WHERE OBJ1.STATUS ='INVALID' AND
      OBJ1.OBJECT_NAME = DEP.NAME AND
      OBJ2.OBJECT_NAME = DEP.REFERENCED_NAME ;
```

□

Les principales tables systèmes décrivant des blocs PL/SQL stockés sont répertoriées dans le tableau suivant.

Nom des vues systèmes	Description	Liste des attributs
USER_OBJECTS	répertorie tous les objets de l'utilisateur, en particulier les blocs stockés. Pour ces derniers, l'attribut OBJECT_TYPE peut prendre les valeurs : FUNCTION, PROCEDURE, PACKAGE, PACKAGE BODY	OBJECT_NAME VARCHAR2(128) SUBOBJECT_NAME VARCHAR2(30) OBJECT_ID NUMBER DATA_OBJECT_ID NUMBER OBJECT_TYPE VARCHAR2(18) CREATED DATE LAST_DDL_TIME DATE TIMESTAMP VARCHAR2(19) STATUS VARCHAR2(7) …
USER_ERRORS	répertorie toutes les erreurs de compilation de tous les blocs utilisateurs stockés.	NAME NOT NULL VARCHAR2(30) TYPE VARCHAR2(12) SEQUENCE NOT NULL NUMBER LINE NOT NULL NUMBER POSITION NOT NULL NUMBER TEXT NOT NULL VARCHAR2(4000)
USER_SOURCE	stocke le code des blocs utilisateurs stockés. Chaque tuple correspond à une ligne de code PL/SQL.	NAME VARCHAR2(30) TYPE VARCHAR2(12) LINE NUMBER TEXT VARCHAR2(4000)
USER_DEPENDENCIES	décrit tous les liens des blocs utilisateurs stockés entre eux et avec les relations de la base.	NAME NOT NULL VARCHAR2(30) TYPE VARCHAR2(12) REFERENCED_OWNER VARCHAR2(30) REFERENCED_NAME VARCHAR2(64) REFERENCED_TYPE VARCHAR2(12) REFERENCED_LINK_NAME VARCHAR2(128)

		SCHEMAID NUMBER DEPENDENCY_TYPE VARCHAR2(4)

Tableau 2 : les tables systèmes d'ORACLE pour les blocs stockés

12 - Quelques références

[1] Steven Feuerstein "Oracle PL/SQL, guide du programmeur", Édition O'REILLY, 2001.

[2] Documentation ORACLE en ligne :http://h50.isi.u-psud.fr/docmiage/oracle/doc/server.817/index.htm
PL/SQL : http://www.csis.gvsu.edu/GeneralInfo/Oracle/appdev.920/a96624/06_ora.htm

[3] http://sanjayahuja.tripod.com/tech/plsql/plsql.html propose un survol (en anglais) de PL/SQL.

[4] http://www.quest.com/whitepapers/PL_SQL_Best_Practices2.pdf
Conseils (en anglais) pour le développement en PL/SQL.

[5] http://www.cs.helsinki.fi/u/laine/info/kevat97/pl_sql.html
Support de cours (en anglais) sur PL/SQL.