Oracle
Le langage procédural PL/SQL

Denis Roegel
roegel@loria.fr
IUT Nancy 2

1998/1999

Sommaire

Ce document est une adaptation d'un chapitre de l'ouvrage *Oracle Unleashed* (SAMS Publishing, 1996).

1 Introduction

Nous détaillons ici les structures de PL/SQL, une interface procédurale au système de gestion de base de données relationnelle Oracle. Les structures de PL/SQL sont similaires à celles des langages évolués et fournissent une méthode souple pour manipuler l'information d'une base de données.

2 Création de *packages*

2.1 Procédures groupées

PL/SQL permet de grouper tous les programmes dans un objet de la base de données appelé un *package*. Un ensemble complet de programmes PL/SQL effectuant une certaine tâche est appelé un *package*.

2.2 Création de *packages*

Avant d'étudier les divers aspects du langage PL/SQL, examinons la syntaxe de création d'un *package* pour l'écriture d'un script facilitant la maintenance en cas de changements. Ce code est la première étape d'un *package* calculant les totaux en dollars dans une commande de marchandises. Cet exemple illustre quelques commandes de création de *packages* et de scripts.

```
set echo on
spool order_total
CREATE OR REPLACE PACKAGE order_total
AS
   (spécifications du package)
END order_total
CREATE OR REPLACE PACKAGE BODY order_total
AS
    (spécifications du corps du package)
END order_total;
DROP PUBLIC SYNONYM order_total;
CREATE PUBLIC SYNONYM order_total for order_total;
GRANT EXECUTE ON order_total TO PUBLIC;
spool off
SELECT *
FROM user_errors
WHERE name='ORDER_TOTAL'
;
```

La première commande dans ce script, SET ECHO ON, affiche un listing du *package* à l'écran comme il a été compilé. La commande ECHO combinée avec la commande SPOOL nom crée un fichier de listing (order_total.lst) à des fins de *debugging*. Ce fichier contiendra la compilation de la procédure, y compris les erreurs, avec les numéros de ligne.

CREATE OR REPLACE PACKAGE nom est la commande qui démarre la construction de la procédure dans la base de données. Les déclarations d'objets et de sous-programmes dans la zone de spécification du *package* sont visibles par vos applications. On peut imaginer cette zone comme l'interface entre les applications et le code PL/SQL ; au minimum, il faut définir la procédure d'entrée ici. En cas de modification à n'importe quelle spécification dans cette zone, il faut recréer les applications. La commande END signifie la fin de la zone de spécification du *package*.

L'instruction CREATE OR REPLACE PACKAGE BODY nom débute la zone de spécification pour la déclaration d'objets PL/SQL et des sous-programmes que seul le *package* peut voir. Cette zone n'est pas visible pour

l'application et n'est pas nécessaire dans la conception de *packages*. Toutefois, le fait de concevoir des *packages* de cette manière vous autorise à modifier les spécifications du corps du *package* sans altérer l'interface de l'application. Par conséquent, les applications n'ont pas besoin d'être recompilées lorsque ces spécifications internes changent. Une fois encore, l'instruction `END` indique la fin des spécifications du corps du *package*.

Dans l'exemple, le nom `order_total` a été choisi à la fois pour le nom du *package* et le nom du corps du *package*, mais les noms auraient pu être différents.

Les trois commandes suivantes sont liées et permettent à tous les utilisateurs d'accéder aux procédures définies dans le *package*. Tout d'abord, tout synonyme public existant est supprimé et puis recréé. La commande `GRANT` fournit un accès « public » au *package*.

`GRANT` est une commande qui doit être exécutée depuis un compte privilégié. À la fin du script se trouve la commande `SPOOL OFF` qui clôt la sortie dans le fichier de listing. Ceci est suivi d'une commande `SELECT` qui affiche toutes les erreurs de compilation dans le terminal où le script a été lancé. Le nom du champ qui apparaît dans cette commande `SELECT` identifie le nom du *package* créé et doit être en majuscules.

Le passage du numéro de ligne dans le message d'erreur à la ligne correspondante dans ou bien le *package* ou bien le corps du *package* dans le fichier de listing va rendre la correction bien plus rapide. Une fois créé, le script peut être exécuté en utilisant une commande `SQL*PLUS` de la manière suivante :

```
sqlplus (username/password) @ot
```

Le login spécifié doit être un compte privilégié. Après le `@` se trouve le nom du script qui contient le texte de création de *package*. Dans ce cas, le nom du script est `ot.sql`, et comme `.sql` est l'extension par défaut des scripts `SQL`, il n'est pas nécessaire de l'inclure sur la ligne de commande `sqlplus`.

3 Création de sous-programmes de *packages*

La création de sous-programmes au sein d'un *package* est l'étape suivante dans le développement d'un *package*. Il faut décider quels programmes seront des programmes d'interface avec une application et quels programmes ne seront disponibles qu'au sein du *package*. Ceci déterminera où la spécification du sous-programme résidera — dans le *package* ou dans le corps du *package*. Il y a deux types de sous-programmes en `PL/SQL`, les procédures et les fonctions.

3.1 Définition d'une procédure

Pour définir une procédure, il faut spécifier un nom de routine et les paramètres qui sont reçus et rendus par la routine. Dans l'exemple de `order_total`, le code qui suit définit la routine d'interface avec une application et réside dans la zone de spécification de *package* :

```
PROCEDURE
  get_order_total (
    in_order_num IN NUMBER,
    out_status_code OUT VARCHAR2,
    out_msg OUT VARCHAR2,
    out_merch_total OUT NUMBER,
    out_shipping IN OUT NUMBER,
    out_taxes IN OUT NUMBER,
    out_grand_total OUT NUMBER
  );
```

Le mot-clé `PROCEDURE` débute la définition de la routine d'interface `get_order_total`. Entre parenthèses se trouvent les paramètres à passer entre l'application et le *package* `order_total`. Le point-virgule marque la fin de la définition de la procédure.

La modularité est la clé d'une conception d'un bon *package*. Si vous limitez la taille des sous-programmes, votre code sera plus facile à concevoir et corriger.

3.2 Définition de fonction

La définition d'une fonction est très analogue à la définition d'une procédure, comme le montre l'exemple ci-dessous :

```
FUNCTION
  calc_ship_charges (
    in_merch_total IN NUMBER
  ) RETURN NUMBER;
```

Le mot-clé `FUNCTION` débute la définition de la fonction de *package* `calc_ship_charges`. Entre parenthèses se trouvent les paramètres à passer à la fonction pour calculer les frais de port. Le mot-clé `RETURN` identifie le type de donnée de la valeur calculée qui va être retournée. Le point-virgule indique la fin de la définition de la fonction.

3.3 Modes des paramètres de sous-programmes

Les paramètres peuvent être définis `IN` (mode par défaut), `IN OUT` ou `OUT`, en fonction de la nature de l'information devant être passée. Le premier paramètre, `in_order_num`, est défini avec le mode `IN`, ce qui le désigne comme une valeur passée au sous-programme. Le fait de définir un paramètre `IN` l'empêche de recevoir une valeur dans la routine.

Les paramètres `out_status_code`, `out_msg`, `out_merch_total` et `out_grand_total` de l'exemple de définition de procédure sont définis comme des valeurs `OUT` rendues à l'appelant. Ces paramètres sont non initialisés à l'entrée de la routine et peuvent recevoir une valeur à l'intérieur de la routine. Le fait de désigner un paramètre `OUT` interdit de l'utiliser dans une expression de sous-programme.

Les paramètres `out_shipping` et `out_taxes` sont définis avec le mode `IN OUT`. Ces paramètres sont des variables initialisées qui sont disponibles pour des réaffectations dans le sous-programme.

3.4 Spécifications de sous-programmes

Après avoir défini un sous-programme et ses paramètres, on développe du code pour le sous-programme. L'exemple suivant illustre quelques constructions de base qu'il faut connaître lorsque l'on code un sous-programme :

```
PROCEDURE
  init_line_items
IS
  (variables locales)
BEGIN
  (corps du sous-programme)
  EXCEPTION
    (traitement des exceptions)
END init_line_items;
```

Dans cet exemple, le nom de la procédure est `init_line_items` avec les variables locales spécifiées après le mot-clé `IS`. Le mot `BEGIN` est le vrai début de la procédure (ou fonction) où le code du sous-programme est développé en même temps que le traitement des exceptions. La procédure est achevée avec `END init_line_items`.

La liste des paramètres de la procédure doit correspondre exactement à la liste des paramètres de la spécification pour la procédure qui est développée. Ceci inclut les types de données et modes des paramètres inclus dans la spécification.

3.5 Paramètres par défaut de la procédure

Afin d'accroître la flexibilité des appels de procédure, des valeurs par défaut peuvent être spécifiées dans la définition de la procédure. De cette manière, la procédure peut être appelée avec tous, un ou aucun des paramètres spécifiés. Des valeurs par défaut sont données aux paramètres qui ne sont pas transmis. L'exemple suivant illustre une définition de procédure utilisant des paramètres par défaut :

```
PROCEDURE
  calc_ship_charges(
    merch_total NUMBER DEFAULT 5.95) IS
  ...
```

Les références à la procédure `calc_ship_charges` peuvent ou non inclure une valeur pour `merch_total`. Sans cette valeur, la valeur par défaut est `5.95` comme indiqué.

3.6 Procédures indépendantes

Les procédures qui ne font pas partie d'un *package* sont appelées *indépendantes* car elles sont définies indépendamment. Ces types de procédures ne sont pas disponibles pour être référencées depuis d'autres outils Oracle. Une autre limitation des procédures indépendantes est qu'elles sont compilées à l'exécution, ce qui ralentit celle-ci.

4 Curseurs

`PL/SQL` utilise des curseurs pour tous les accès à des informations de la base de données. Le langage supporte à la fois l'emploi de curseurs implicites et explicites. Les curseurs implicites sont ceux qui sont établis lorsqu'un curseur explicite n'a pas été déclaré. Il faut utiliser des curseurs explicites ou des curseurs de boucles `FOR` dans toutes les requêtes qui renvoient plusieurs lignes.

4.1 Déclaration de curseurs

Les curseurs sont définis dans la zone des variables de sous-programmes `PL/SQL` en utilisant l'instruction `CURSOR name IS`, comme montré dans l'exemple suivant :

```
CURSOR c_line_item IS
(instruction sql)
```

L'instruction `SQL` peut être n'importe quelle requête valide. Après l'initialisation d'un curseur, les actions d'un curseur peuvent être contrôlées avec les instructions `OPEN`, `FETCH` et `CLOSE`.

4.2 Le contrôle d'un curseur

Pour utiliser un curseur afin de manipuler des données, il faut utiliser l'instruction `OPEN name` pour exécuter la requête et identifier toutes les lignes qui satisfont le critère de sélection. Les extractions ultérieures de lignes sont réalisées avec l'instruction `FETCH`. Lorsque toutes les données sont traitées, l'instruction `CLOSE` clôt toute activité associée avec le curseur ouvert. Ce qui suit est un exemple de contrôle de curseur :

```
OPEN c_line_item;
  ...
    FETCH c_line_item
      INTO li_info;
  ...
  (traitement de la ligne extraite)
  ...
CLOSE c_line_item;
```

Ce code ouvre le curseur `c_line_item` et traite les lignes extraites. Après l'extraction et le traitement de toute l'information, le curseur est fermé. Le traitement des lignes extraites est typiquement contrôlé par des itérations de boucles comme discuté plus loin.

4.3 Attributs des curseurs explicites

Il y a quatre attributs associés aux curseurs `PL/SQL`.

- `%NOTFOUND`
- `%FOUND`
- `%ROWCOUNT`
- `%ISOPEN`

Tous les attributs de curseur s'évaluent à `TRUE`, `FALSE` ou `NULL`, en fonction de la situation. L'attribut `%NOTFOUND` s'évalue à `FALSE` quand une ligne est extraite, `TRUE` si le dernier `FETCH` n'a pas renvoyé une valeur et `NULL` si le curseur `SELECT` n'a pas renvoyé de données. L'attribut `%FOUND` est l'opposé logique de `%NOTFOUND` par rapport à `TRUE` et `FALSE`, mais s'évalue néanmoins à `NULL` si le curseur ne renvoie pas de données. `%ROWCOUNT` peut être utilisé pour déterminer combien de rangées ont été sélectionnées à un moment donné dans le `FETCH`. Cet attribut est incrémenté après la sélection réussie d'une ligne. De plus, `%ROWCOUNT` est à zéro quand le curseur est ouvert pour la première fois. Le dernier attribut, `%ISOPEN`, est ou bien `TRUE` ou bien `FALSE`, suivant que le curseur associé est ouvert ou non. Avant que le curseur ne soit ouvert et après qu'il soit fermé, `%ISOPEN` vaut `FALSE`. Dans les autres cas, cet attribut s'évalue à `TRUE`.

4.4 Paramètres des curseurs

On peut spécifier des paramètres pour les curseurs de la même manière que pour des sous-programmes. L'exemple suivant illustre la syntaxe de déclaration de curseurs avec des paramètres :

```
CURSOR c_line_item (order_num IN NUMBER) IS
  SELECT merch_gross, recipient_num
  FROM line_item
  WHERE order_num = g_order_num;
```

Le mode des paramètres est toujours `IN`, mais les types de données peuvent être n'importe quels types de données valides. Un paramètre de curseur ne peut être référencé que pendant la requête déclarée.

La flexibilité au sein des paramètres de curseurs permet au développeur de passer différents nombres de paramètres à un curseur en utilisant le mécanisme des paramètres par défaut. Ceci est illustré dans l'exemple ci-dessous :

```
CURSOR c_line_item
  (order_num INTEGER DEFAULT 100,
    line_num INTEGER DEFAULT 1) IS ...
```

En utilisant la déclaration `INTEGER DEFAULT`, on peut passer tous, un, ou aucun des paramètres de ce curseur en fonction du code appelant.

4.5 Création de *packages* de curseurs

Un *package* de curseurs est similaire à un *package* de procédures en ce que l'on spécifie le curseur et son attribut de retour, `%TYPE` or `%ROWTYPE`, dans la zone de spécification du *package*. On spécifie ensuite le corps du curseur dans la zone de spécification du corps du *package*. Le fait de regrouper un curseur de cette manière donne la flexibilité de changer le corps du curseur sans avoir à recompiler les applications qui font référence à la procédure groupée. Ce qui suit est un exemple de *package* de curseur :

```
CREATE OR REPLACE PACKAGE order_total
AS
```

```
CURSOR c_line_item RETURN line_item.merch_gross%TYPE;
   ...
END order_total;
CREATE OR REPLACE PACKAGE BODY order_total
AS
  CURSOR c_line_item RETURN line_item.merch_gross%TYPE
    SELECT merch_gross
    FROM line_item
    WHERE order_num = g_order_num;
  ...
END order_total;
```

Dans cet exemple, la variable rendue est de même type que `line_item.item_merch_gross`. On peut utiliser l'attribut `%ROWTYPE` pour spécifier un enregistrement `RETURN` qui reflète une ligne dans une table de la base de données.

5 Variables de procédures

Un aspect important d'un langage est la manière de définir les variables. Une fois que les variables sont définies, `PL/SQL` permet de les utiliser dans des commandes `SQL` ou dans d'autres commandes du langage. La définition de constantes au sein de `PL/SQL` suit les mêmes règles. De même, on peut définir des variables et constantes locales à un sous-programme ou globales à un *package* qui est créé.

Il faut définir les variables et les constantes avant de les référencer dans une autre construction.

5.1 Déclaration de variables

Tout type de donnée de `PL/SQL` ou `SQL` est un type valide dans une définition de variable. Les types de données les plus utilisés sont `VARCHAR2`, `DATE`, `NUMBER` (types de `SQL`), `BOOLEAN` et `BINARY_INTEGER` (types de `PL/SQL`). Les types de données scalaires et composés de `PL/SQL` sont discutés de manière plus détaillée plus loin.

5.2 Variables locales

Supposons que l'on veuille déclarer deux variables locales nommées `merch_gross` et `recip_count`. La première, `merch_gross`, va contenir un nombre flottant à 10 chiffres arrondi à deux décimales ; `recip_count` va contenir un compteur entier. Ces variables sont déclarées de la manière suivante :

```
merch_gross NUMBER;
recip_count BINARY_INTEGER;
```

On peut aussi déclarer `merch_gross` dans cet exemple avec `NUMBER(10,2)` pour expliciter le nombre total de chiffres et l'arrondi. Toutefois, si une telle déclaration est liée à un champ de la base de données, elle doit changer lorsque la définition de la base de données change.

On peut utiliser deux méthodes pour donner des valeurs aux variables. La première est d'utiliser un opérateur d'affectation comme suit :

```
merch_gross := 10.50;
```

La seconde méthode est d'utiliser une commande `SQL SELECT` ou `FETCH` qui définit une valeur de la base de données comme suit :

```
SELECT merch_gross
INTO merch_gross
FROM line_item
WHERE order_num = g_order_num;
```

5.3 Constantes locales

La déclaration d'une constante est similaire à la déclaration d'une variable sauf que le mot clé `CONSTANT` doit suivre le nom de la « variable ». Il faut immédiatement donner une valeur à la constante.

```
tax_rate CONSTANT NUMBER := 0.03;
```

5.4 Variables globales

Les variables globales sont définies de la même manière que des variables locales, mais elles sont définies en dehors de toute définition de procédure. Supposons que l'on veuille définir les variables `g_order_num` et `g_recip_counter` pour qu'elles soient accessibles depuis tous les sous-programmes du *package*. Cela peut se faire ainsi :

```
CREATE OR REPLACE PACKAGE BODY
  order_total
AS
  ...
g_order_num NUMBER;
g_recip_counter BINARY_INTEGER;
  ...
PROCEDURE
  ...
```

Il faut noter que ces variables globales sont définies dans la zone de spécification du corps du *package* pour éviter qu'elles ne soient « vues » par des applications qui appellent la procédure groupée `order_total`.

Si l'on utilise des noms de variables identiques à des noms de colonnes de la base de données, les résultats d'opérations `SELECT` ou `UPDATE` impliquant ces variables sont imprévisibles.

5.5 Mot-clé `DEFAULT`

Le mot-clé `DEFAULT` permet d'initialiser des variables sans utiliser l'opérateur d'affectation comme dans l'exemple suivant :

```
merch_gross NUMBER DEFAULT 10.50;
```

On peut aussi utiliser le mot-clé `DEFAULT` pour initialiser les paramètres d'un curseur dans un sous-programme ou des champs dans un enregistrement défini par l'utilisateur.

5.6 Attributs des variables et constantes

Les deux attributs des variables et constantes `PL/SQL` sont `%TYPE` et `%ROWTYPE`. L'attribut `%TYPE` permet de déclarer des variables similaires à des colonnes de la base de données sans connaître le type de la colonne. `merch_gross` peut être défini de la manière suivante :

```
merch_gross line_item.merch_gross%TYPE;
```

La définition d'une variable de cette manière permet de rendre effectifs des changements à la base de données lors de la prochaine compilation sans changer le code.

L'attribut `%ROWTYPE` permet de représenter une ligne d'une table avec un type de donnée enregistrement qui masque les colonnes de la base de données. Considérons l'échantillon de données dans la table `LINE_ITEM` ci-dessous :

Nom de la colonne	`order_num`	`line_num`	`merch_gross`	`recipient_num`
Donnée	100	1	10.50	1000

Un curseur peut être défini au sein d'une procédure (voir plus haut) afin d'extraire des informations de la table LINE_ITEM. En même temps que le curseur, on définit une variable ROWTYPE pour stocker les champs de cette ligne comme suit :

```
CURSOR c_line_item IS
SELECT merch_gross, recipient_num
FROM line_item
WHERE order_num = g_ordnum;
li_info c_line_item%ROWTYPE;
```

Pour extraire les données, on utilise FETCH :

```
FETCH c_line_item
INTO li_info;
```

Après le FETCH, on utilise la notation « . » pour accéder à l'information extraite de la base de données.

```
g_order_merch_total := g_order_merch_total + li_info.merch_gross;
```

6 Types de données scalaires

PL/SQL supporte une vaste gamme de types de données scalaires pour définir des variables et des constantes. À la différence des types de données composites, les types de données scalaires n'ont pas de composantes accessibles. Ces types de données tombent dans l'une des catégories suivantes :

- Booléen
- Date/heure
- Caractère
- Nombre

Chaque catégorie va maintenant être étudiée de plus près.

6.1 Booléen

Le type de données BOOLEAN, qui ne prend pas de paramètres, est utilisé pour stocker une valeur binaire, TRUE ou FALSE. Ce type de donnée peut aussi stocker la « non-valeur » NULL.

6.2 Date/Heure

Le type de données DATE, qui ne prend aucun paramètre, est utilisé pour stocker des valeurs de dates. Ces valeurs de date incluent l'heure lorsqu'elles sont stockées dans une colonne de la base de données. Les dates peuvent s'étendre du 1er janvier 4712 av. J.-C. au 31 décembre 4712. Les valeurs par défaut pour le type de données DATE sont les suivantes :

- Date : premier jour du mois courant
- Heure : minuit

6.3 Caractère

Le type de données caractère inclut CHAR, VARCHAR2, LONG, RAW et LONG RAW. CHAR est destiné aux données de type caractère de longueur fixe et VARCHAR2 stocke des données de type caractère de longueur variable. LONG stocke des chaînes de longueur variable ; RAW et LONG RAW stockent des données binaires ou des chaînes d'octets. Les types de données CHAR, VARCHAR2 et RAW prennent un paramètre optionnel pour spécifier la longueur.

```
type(max_len)
```

Ce paramètre de longueur, `max_len`, doit être un entier littéral et non une constante ou une variable. La table ci-dessous montre les longueurs maximales et les largeurs des colonnes de la base pour les types de données caractère.

Type de données	CHAR	VARCHAR2	LONG	RAW	LONG RAW
Longueur maximale	32767	32767	32760	32767	32760
Largeur maximale de colonne	255	2000	2147483647	255	2147483647

Avec cette table, on peut voir la contrainte d'introduction de données `CHAR`, `VARCHAR2` et `RAW` dans des colonnes de même type de la base de données. La limite est la largeur de la colonne. Toutefois, on peut insérer des données de type `LONG` et `LONG RAW` de n'importe quelle longueur dans des colonnes similaires parce que la largeur de colonne est bien plus grande.

6.4 Nombre

Il y a deux types de données dans la catégorie des nombres : `BINARY_INTEGER` et `NUMBER`. `BINARY_INTEGER` stocke des entiers signés sur l'étendue de -2^{31} à $2^{31}-1$. L'utilisation la plus courante de ce type de donnée est celle d'un index pour des tables `PL/SQL`. Le stockage de nombres de taille fixe ou flottants de n'importe quelle taille est possible avec le type de donnée `NUMBER`. Pour des nombres flottants, la précision et l'échelle peuvent être spécifiés avec le format suivant :

```
NUMBER(10,2)
```

Une variable déclarée de cette manière a un maximum de 10 chiffres et l'arrondi se fait à deux décimales. La précision par défaut est le plus grand entier supporté par le système et 0 est l'échelle par défaut. L'intervalle de précision va de 1 à 38 alors que l'échelle va de -84 à 127.

7 Types de données composés

Les deux types de données composites de `PL/SQL` sont `TABLE` et `RECORD`. Le type de donnée `TABLE` permet à l'utilisateur de définir un tableau `PL/SQL`. Le type de données `RECORD` permet d'aller au-delà de l'attribut de variable `%ROWTYPE` ; avec ce type, on peut spécifier des champs définis par l'utilisateur et des types de données pour ces champs.

7.1 Traitement des tableaux

Le type de données composé `TABLE` donne au développeur un mécanisme pour traiter les tableaux. Bien que ce type soit limité à une colonne d'information par tableau `PL/SQL`, on peut stocker n'importe quel nombre de lignes pour cette colonne. Les versions ultérieures d'Oracle offriront plus de flexibilité dans l'emploi des tableaux.

Ce qui suit illustre comment on peut définir un tableau `PL/SQL` nommée `g_recip_list` (l'information sera utilisée globalement) dans l'exemple de `order_total`.

```
TYPE RecipientTabTyp IS TABLE OF NUMBER(22)
  INDEX BY BINARY_INTEGER;
...
g_recip_list RecipientTabTyp;
```

Pour initialiser un tableau, il faut tout d'abord définir un nom de tableau ou un type. Dans l'exemple précédent, c'est `RecipientTabTyp`. Cette colonne de tableau est définie comme un nombre avec au maximum 22 chiffres. La colonne peut être définie avec n'importe quel type de données valide de `PL/SQL` ; toutefois, la clé primaire, ou `INDEX`, doit être de type `BINARY_INTEGER`. Après avoir défini la structure du tableau, elle peut être utilisée dans des définitions de variables, comme c'est le cas pour `RecipientTabTyp` dans l'exemple précédent.

7.2 Constructions de tableaux

Après leur initialisation, les tableaux sont disponibles pour le stockage d'information. Pour stocker de l'information dans le tableau `g_recip_list` défini dans l'exemple précédent, il suffit de référencer le tableau avec une valeur numérique. Cela est illustré dans l'exemple suivant :

```
g_recip_list(j) := g_recipient_num(i)
```

Dans cet exemple, `i` et `j` sont des compteurs avec les valeurs $1..n$. À partir du moment où de l'information est stockée dans un tableau, on peut y accéder, de manière numérique, comme indiqué dans l'exemple. Dans ce cas, les lignes de `g_recipient_num` sont référencées pour stockage dans `g_recip_list`.

Le fait de référencer une ligne non initialisée dans un tableau `PL/SQL` cause une erreur `NO_DATA_FOUND` (voir la section sur le traitement des exceptions plus loin).

7.3 Traitement des enregistrements

Le type de donnée composite `RECORD` procure au développeur un mécanisme pour traiter les enregistrements comme décrit précédemment. Bien que l'on ne puisse pas initialiser un tableau au moment de sa déclaration, il est possible de le faire avec des enregistrements, comme illustré dans l'exemple suivant :

```
TYPE LineRecTyp IS RECORD
  (merch_gross NUMBER := 0,
    recip_num NUMBER := 0 );
  ...
li_info LineRecTyp;
```

La définition d'un enregistrement de type `LineRecTyp` permet des déclarations telles que `li_info` de ce type. Cette méthode de déclaration d'enregistrement peut être utilisée à la place de la déclaration `li_info` dans l'exemple `%ROWTYPE` précédent. Tout comme avec `%ROWTYPE`, les références aux données des enregistrements se font avec la notation « `.` ».

```
g_order_merch_total := g_order_merch_total + li_info.merch_gross;
```

Il y a trois moyens de donner une valeur à un enregistrement. Tout d'abord, une valeur peut être donnée à un champ d'un enregistrement tout comme on donne une valeur à n'importe quelle variable.

```
li_info.merch_gross := 10.50;
```

Une seconde méthode est de donner une valeur à tous les champs à la fois en utilisant deux enregistrements qui sont déclarés de même type. Supposons que `new_li_info` est une seconde variable de type `LineRecTyp` :

```
new_li_info := li_info;
```

Cette instruction donne à tous les champs de `new_li_info` les valeurs des mêmes champs de `li_info`.

Il n'est pas possible d'affecter des valeurs d'enregistrements de différents types entre eux.

Une troisième manière de donner des valeurs aux champs d'un enregistrement consiste à utiliser les instructions `SQL SELECT` ou `FETCH`.

```
OPEN c_line_item;
  ...
    FETCH c_line_item
    INTO li_info;
```

Dans ce cas, tous les champs de `li_info` reçoivent des valeurs provenant des informations extraites par la commande `FETCH` sur le curseur `c_line_item`.

8 Structures de contrôle

Tout langage procédural a des structures de contrôle qui permettent de traiter l'information d'une manière logique en contrôlant le flot des informations. Les structures disponibles au sein de PL/SQL incluent IF-THEN-ELSE, LOOP et EXIT-WHEN. Ces structures procurent de la flexibilité dans la manipulation des données de la base de données.

8.1 Boucles

L'utilisation de la commande LOOP fournit un traitement itératif basé sur des choix logiques. La construction de base des boucles « LOOP » est montrée dans l'exemple suivant :

```
<<nom>>
LOOP
  (traitement répétitif)
END LOOP nom;
```

Pour sortir d'une boucle de ce genre, il faut une commande EXIT ou GOTO basée sur une condition du traitement. En cas de levée d'exception définie par l'utilisateur, la boucle LOOP s'achève aussi. Examinons maintenant trois types de boucles PL/SQL qui définissent des conditions explicites de terminaison.

Une boucle peut être nommée comme cela a été montré dans l'exemple en utilisant une étiquette telle que <<nom>> juste avant l'instruction LOOP. Bien que ce ne soit pas obligatoire, l'étiquetage permet de garder une meilleure trace de l'imbrication des boucles.

8.2 Boucles WHILE

La boucle WHILE vérifie l'état d'une expression PL/SQL qui doit s'évaluer à TRUE, FALSE ou NULL au début de chaque cycle de traitement. Ce qui suit est un exemple d'utilisation de boucles WHILE :

```
WHILE (expression) LOOP
  (traitement de boucle)
END LOOP;
```

Comme indiqué, le programme évalue l'expression au début de chaque cycle de boucle. Le programme exécute le traitement de la boucle si l'expression s'évalue à TRUE. Une valeur FALSE ou NULL termine la boucle. Les itérations à travers la boucle sont exclusivement déterminées par l'évaluation de l'expression.

8.3 Boucles FOR numériques

Les itérations de boucles peuvent être contrôlées avec des boucles FOR numériques. Ce mécanisme permet au développeur d'établir un intervalle d'entiers pour lesquels la boucle va être itérée. L'exemple suivant du *package* order_total illustre les boucles numériques FOR :

```
<<recip_list>>
  FOR i in 1..g_line_counter LOOP
    (traitement de boucle)
  END LOOP recip_list;
```

Dans cet exemple, la boucle est itérée pour les entiers de 1 jusqu'à la valeur de g_line_counter. La valeur de l'index de boucle i est vérifiée au début de la boucle et incrémentée à la fin de la boucle. Lorsque i est égal à g_line_counter + 1, la boucle termine.

8.4 Boucles FOR de curseurs

Les boucles FOR de curseurs combinent le contrôle de curseurs et des structures de contrôle supplémentaires pour la manipulation d'informations de bases de données. L'index de boucle, l'ouverture de curseur, le FETCH et la fermeture de curseur sont tous implicites lorsque l'on utilise des boucles FOR de curseurs. Considérons l'exemple suivant :

```
CURSOR c_line_item IS
(instruction sql)
BEGIN
  FOR li_info IN c_line_item LOOP
    (traitement de l'enregistrement extrait)
  END LOOP;
END;
```

Comme montré, le programme déclare explicitement le curseur c_line_item avant qu'il ne soit référencé dans la boucle FOR. Lorsque le programme pénètre dans la boucle FOR, le code ouvre implicitement c_line_item et crée implicitement l'enregistrement li_info comme si la déclaration suivante était présente :

```
li_info c_line_item%ROWTYPE;
```

Dès l'entrée dans la boucle, le programme peut référencer les champs de l'enregistrement li_info qui reçoivent des valeurs par le FETCH implicite à l'intérieur de la boucle FOR. Les champs de li_info reflètent la ligne extraite par le curseur c_line_item.

Lorsqu'il n'y a plus de données pour FETCH, le curseur c_line_item est implicitement fermé.

Il n'est pas possible de référencer l'information contenue dans li_info en dehors de la boucle de curseur.

8.5 Structure de contrôle conditionnelle

La structure IF-THEN-ELSE permet d'avoir des traitements qui dépendent de certaines conditions. Par exemple, considérons des commandes de marchandises avec des éléments sur plusieurs lignes où une liste de destinataires est construite. En utilisant des structures de contrôle conditionnelles et itératives pour construire la liste des destinataires, le code est le suivant :

```
PROCEDURE
  init_recip_list
IS
  recipient_num NUMBER;
  i BINARY_INTEGER;
  j BINARY_INTEGER := 1;
  k BINARY_INTEGER;
  BEGIN
    g_out_msg := 'init_recip_list';
    <<recip_list>>
    FOR i in 1..g_line_counter LOOP
      IF i = 1 THEN
        g_recip_list(j) := g_recipient_num(i);
        j := j + 1;
        g_recip_list(j) := 0;
      ELSE
        FOR k in 1..j LOOP
          IF g_recipient_num(i) = g_recip_list(k) THEN
             exit;
          ELSIF k = j THEN
            g_recip_list(j) := g_recipient_num(i);
            j := j + 1;
```

```
            g_recip_list(j) := 0;
          end IF;
        end LOOP;
      end IF;
    end LOOP recip_list;
  END;
```

Dans l'exemple `order_total`, le sous-programme `init_recip_list` construit une liste de numéros de destinataires uniques pour le calcul des frais de port supplémentaires. Il y a une boucle de contrôle `FOR` qui parcourt chaque numéro de destinataire trouvé sur une commande particulière. Le tableau `g_recip_list` est initialisé avec le premier numéro de destinataire et les numéros suivants sont comparés avec tous les numéros uniques dans `g_recip_list` jusqu'à ce qu'une liste de tous les destinataires ait été rassemblée.

Cet exemple illustre aussi l'extension `ELSIF` de `IF-THEN-ELSE`. Cette partie fournit une structure de contrôle supplémentaire avec des test de contraintes additionnelles. L'emploi de `ELSIF` requiert aussi un `THEN`.

Un autre exemple est l'emploi de `EXIT-WHEN` qui permet la complétion d'une boucle lorsque certaines conditions sont satisfaites. Considérons l'exemple de sortie de boucle `FETCH` suivant :

```
open c_line_item;
  loop
    fetch c_line_item
    into li_info;
    EXIT WHEN (c_line_item%NOTFOUND) or (c_line_item%NOTFOUND is NULL);
```

Dans cet exemple, la boucle est terminée lorsqu'on ne trouve plus de données pour satisfaire le `SELECT` du curseur `c_line_item`.

L'emploi de `%NOTFOUND` ou `%FOUND` peut causer des boucles infinies si l'on ne vérifie pas que ces attributs sont évalués à `NULL` dans un test logique `EXIT-WHEN`.

9 Traitement des exceptions

Le traitement des exceptions `PL/SQL` est un mécanisme pour manipuler les erreurs rencontrées lors de l'exécution. L'utilisation de ce mécanisme permet à l'exécution de continuer si l'erreur n'est pas suffisamment importante pour produire la terminaison de la procédure. La décision de permettre à une procédure de continuer après une condition d'erreur est une décision que le développpeur doit faire en fonction des erreurs possibles.

Il faut définir le *handler* d'exception au sein de la spécification d'un sous-programme. Les erreurs conduisent le programme à lever une exception et transfèrent le contrôle au *handler* d'exceptions. Après l'exécution du *handler*, le contrôle retourne au bloc dans lequel le handler a été défini. S'il n'y a plus d'instructions exécutables dans le bloc, le contrôle retourne au code appelant.

9.1 Exceptions définies par l'utilisateur

`PL/SQL` permet à l'utilisateur de définir des *handler* d'exceptions dans la zone des déclarations ou des spécifications de sous-programmes. Cela se fait en donnant un nom à l'exception comme dans l'exemple suivant :

```
ot_failure EXCEPTION;
```

Dans ce cas, le nom de l'exception est `ot_failure`. Le code associé au *handler* est écrit dans la zone de spécification `EXCEPTION` comme indiqué ci-dessous :

```
EXCEPTION
  when OT_FAILURE then
    out_status_code := g_out_status_code;
    out_msg := g_out_msg;
```

Cette exception est définie dans l'exemple `order_total` pour capturer l'état et les données associées pour les exceptions `NO_DATA_FOUND` rencontrées dans un sous-programme. Ce qui suit est un exemple d'une exception de sous-programme :

```
EXCEPTION
  when NO_DATA_FOUND then
    g_out_status_code := 'FAIL';
    RAISE ot_failure;
```

Au sein de cette exception se trouve la commande `RAISE` qui transfère le contrôle au *handler* d'exception `ot_failure`. Cette technique pour provoquer des exceptions est utilisée pour invoquer toutes les exceptions définies par l'utilisateur.

9.2 Exceptions définies par le système

Des exceptions internes à `PL/SQL` sont levées automatiquement en cas d'erreur. Dans le précédent exemple, `NO_DATA_FOUND` est une exception définie par le système. La table suivante est une liste complète des exceptions internes.

Nom de l'exception	Erreur Oracle
CURSOR_ALREADY_OPEN	ORA-06511
DUP_VAL_ON_INDEX	ORA-00001
INVALID_CURSOR	ORA-01001
INVALID_NUMBER	ORA-01722
LOGIN_DENIED	ORA-01017
NO_DATA_FOUND	ORA-01403
NOT_LOGGED_ON	ORA-01012
PROGRAM_ERROR	ORA-06501
STORAGE_ERROR	ORA-06500
TIMEOUT_ON_RESOURCE	ORA-00051
TOO_MANY_ROWS	ORA-01422
TRANSACTION_BACKED_OUT	ORA-00061
VALUE_ERROR	ORA-06502
ZERO_DIVIDE	ORA-01476

Outre cette liste d'exceptions, il y a une exception « attrape-tout » nommée `OTHERS` qui permet d'attraper toutes les erreurs pour lesquelles il n'y a pas de traitement d'erreur spécifique. Cette exception est illustrée sur l'exemple suivant :

```
when OTHERS then
  out_status_code := 'FAIL';
  out_msg := g_out_msg || ' ' || SUBSTR(SQLERRM, 1, 60);
```

Cette technique est utilisée dans l'exemple de la procédure `order_total` pour piéger toutes les erreurs de procédure autres que `NO_DATA_FOUND`. L'information renvoyée au code appelant dans `out_msg` est le nom du sous-programme contenu dans `g_out_msg` concaténé avec les 60 premiers caractères retournés par la fonction `SUBSTR` appliquée à la fonction `SQLERRM`.

À la fois `SQLERRM` et `SUBSTR` sont des fonctions internes `PL/SQL`. Une liste complète des fonctions internes est donnée plus loin.

`SQLERRM` ne renvoie un message valide que lorsque cette fonction est appelée au sein d'un *handler* d'exception à moins qu'un argument ne soit passé à la fonction et que cet argument soit un numéro d'erreur valide de `SQL`. Le code d'erreur Oracle est la première partie du message retourné par `SQLERRM`. Ensuite vient le texte associé avec ce code d'erreur.

De cette manière, toutes les erreurs rencontrées pendant l'exécution d'une procédure sont attrapées et repassées à l'application à des fins de correction. Ce qui suit est un exemple d'erreur renvoyée par la procédure `order_total` :

```
FAIL: init_line_items ORA-01001: invalid cursor
```

Ce message d'erreur (formaté par l'application) révèle une opération illégale sur un curseur dans le sous-programme `init_line_items`. La portion du message retournée par `SQLERRM` commence avec le code d'erreur `SQL ORA-01001`. Un autre message d'erreur est illustré dans l'exemple suivant :

```
FAIL: calc_ship_charges
```

Dans ce cas, le sous-programme `calc_ship_charges` avait une erreur `NO_DATA_FOUND`. Ceci est déterminé par le fait qu'aucun message d'erreur `SQL` n'est concaténé avec le texte du message.

10 Commentaires

Bien que certaines personnes pensent que commenter un programme n'est pas nécessaire, il y a deux moyens de mettre des commentaires au sein de procédures `PL/SQL`. La première est destinée à des commentaires sur une seule ligne et la syntaxe est indiquée sur l'exemple suivant :

```
--*************** CRÉATION DU PACKAGE ORDER_TOTALING ***************
```

Un double tiret au début de la ligne indique que cette ligne est un commentaire. Le second moyen est utilisé pour placer une suite de commentaires dans un *package* `PL/SQL`.

```
/* Le code qui suit génère une liste de numéros de destinataires
uniques pour tous les numéros de destinataires pour un certain ordre */
```

Un bloc de commentaires tel que celui-là commence avec `/*` et s'achève avec `*/`. Les commentaires quels qu'ils soient peuvent être placés à n'importe quel endroit du code `PL/SQL`.

Les blocs `PL/SQL` qui sont compilés dynamiquement dans les applications *Oracle Precompiler* ne peuvent pas utiliser les commentaires mono-lignes.

11 Procédures cataloguées

Il est possible de stocker du code `PL/SQL` dans la base Oracle avec les extensions procédurales. Les avantages des procédures cataloguées incluent une maintenance plus aisée, des applications plus petites, une exécution plus rapide et de plus grandes économies de mémoire, pour n'en citer que quelques uns.

11.1 Référencer des procédures cataloguées

Un autre avantage important lié à l'utilisation de procédures cataloguées est la possibilité de référencer la procédure depuis de nombreuses applications Oracle différentes. Il est possible de faire référence à des procédures cataloguées depuis d'autres procédures cataloguées, depuis des *triggers* de bases de données, depuis des applications construites avec des précompilateurs Oracle ou des outils Oracle tels que `SQL*Forms`. L'exemple suivant appelle le *package* `order_total` depuis une autre procédure :

```
order_total.get_order_total (order_num,
                             status_code,
                             message,
                             merch_gross,
                             shipping,
                             taxes,
                             grand_total);
```

L'exemple suivant montre la même procédure `order_total` référencée depuis `PRO*C`, une application du précompilateur Oracle.

```
EXEC SQL
  BEGIN
```

```
    order_total.get_order_total ( :order_num,
                                  :status_code,
                                  :message,
                                  :merch_gross,
                                  :shipping,
                                  :taxes,
                                  :grand_total);
  END;
END-EXEC;
```

Tous les paramètres dans cet exemple de la procédure `order_total` sont des variables liées d'Oracle qu'il faut déclarer avant de les référencer dans le *package*. L'exemple final illustre un appel au *package* `order_total` depuis une application `SQL*Forms`.

```
BEGIN
  ...
  order_total.get_order_total ( order_num,
                                status_code,
                                message,
                                merch_gross,
                                shipping,
                                taxes,
                                grand_total);
  ...
END;
```

Une fois de plus, il faut déclarer toutes les variables passées en paramètre avant d'appeler la procédure.

11.2 États des procédures cataloguées

Après compilation, une procédure cataloguée existe ou bien sous une forme valide ou bien sous une forme non valide. S'il n'y a pas eu de changements à la procédure, elle est considérée valide et peut être référencée. Si un sous-programme ou un objet référencé au sein de la procédure change, son état devient non valide. Seules des procédures dans un état valide sont disponibles pour référence.

Le fait de référencer une procédure qui n'est pas valide entraine la recompilation par Oracle des objets appelés par la procédure référencée. Si la recompilation échoue, Oracle renvoie une erreur d'exécution à l'appelant et la procédure reste dans un état non valide. Sinon, Oracle recompile la procédure référencée et, si la recompilation réussit, l'exécution se poursuit.

11.3 Surcharge

Le concept de surcharge dans `PL/SQL` est lié à l'idée que l'on peut définir des procédures et des fonctions avec le même nom. `PL/SQL` ne considère pas seulement le nom référencé pour déterminer les procédures et fonctions appelées, mais aussi le nombre et le type des paramètres formels.

`PL/SQL` essaie aussi de résoudre les appels de procédures ou de fonctions dans des *packages* définis localement avant de regarder dans des fonctions internes ou dans des *packages* définis globalement. Pour assurer encore davantage l'appel de la bonne procédure, le point (`.`) peut être utilisé comme cela a été illustré dans des exemples antérieurs avec des références à des procédures cataloguées. Le fait de préfixer un nom de procédure ou de fonction par le nom du *package* détermine de manière unique toute référence à une procédure ou une fonction.

12 *Commits*

La commande `COMMIT` est utilisable dans les procédures `PL/SQL` à moins que la procédure soit appelée depuis une application `SQL*Forms`. Pour autoriser les « *commits* » au sein d'une procédure appelée par une

application SQL*Forms, il faut exécuter la commande ALTER SESSION ENABLE COMMIT IN PROCEDURE avant d'invoquer l'objet PL/SQL. Comme cette commande ne peut pas être appelée depuis SQL*Forms, il faut créer une sortie utilisateur depuis laquelle la commande ALTER SESSION peut être exécutée et ensuite appeler la procédure. Ce qui suit est un exemple d'appel de la procédure order_total depuis SQL*Forms au travers d'une sortie utilisateur :

```
user_exit('order_totl');
```

Dans ce cas, la routine order_totl de la sortie utilisateur SQL*Forms fait référence à la procédure groupée order_total.

13 *Package* STANDARD

PL/SQL fournit divers outils dans un *package* appelé STANDARD pour l'utilisation par les développeurs. Ces outils incluent des fonctions internes et des exceptions internes.

13.1 Références à des fonctions internes

Les fonctions PL/SQL internes sont un bon exemple de surcharge par rapport aux noms des procédures et fonctions. Il faut se rappeler que PL/SQL détermine quelle procédure ou fonction est appelée en cherchant une correspondance à la fois avec le nombre et le type des paramètres formels et pas seulement avec le nom de la procédure ou fonction référencée. Considérons les deux fonctions internes appelées TO_NUMBER dans l'exemple suivant :

```
function TO_NUMBER (str CHAR [, fmt VARCHAR2, [, nlsparms] ]) return NUMBER

function TO_NUMBER (str VARCHAR2 [, fmt VARCHAR2 [, nlsparms] ]) return NUMBER
```

Les deux fonctions sont nommées TO_NUMBER, mais le type de données du premier paramètre est CHAR dans la première définition et VARCHAR2 dans la seconde. Les paramètres optionnels sont les mêmes dans les deux cas. PL/SQL résout l'ambiguïté d'un appel à la fonction TO_NUMBER en regardant le type de donnée du premier paramètre. On pourrait aussi avoir une procédure ou fonction définie par l'utilisateur et appelée TO_NUMBER. Dans ce cas, la définition locale prend le pas sur la définition interne de la fonction. Il reste possible d'accéder à la fonction interne en utilisant la notation « . » comme suit :

```
STANDARD.TO_NUMBER ...
```

13.2 Fonctions internes

La fonction TO_NUMBER est un exemple de fonction PL/SQL interne. La table ci-dessous donne une liste complète des catégories de fonctions PL/SQL avec les valeurs de retour par défaut.

Catégorie	Caractère	Conversion	Date	Divers	Nombre
Valeur de retour usuelle	VARCHAR2	Aucune	DATE	Aucune	NUMBER

13.2.1 Fonctions de chaînes

Bien que la plupart des fonctions de chaînes renvoient une valeur de type VARCHAR2, quelques fonctions renvoient d'autres valeurs. La table ci-dessous donne la liste des fonctions de chaînes disponibles avec une brève description, la liste des arguments et la valeur de retour si elle est différente de la valeur de retour la plus probable pour cette catégorie de fonctions. Les arguments optionnels sont indiqués entre crochets. Toutes les fonctions de chaînes internes prennent la forme suivante :

```
function ASCII (char VARCHAR2) return VARCHAR2
```

Fonction	Description	Argument(s)	Valeur de retour
ASCII	Retourne le code standard d'un caractère.	char VARCHAR2	NUMBER
CHR	Retourne le caractère correspondant à un code.	num NUMBER	
CONCAT	Retourne str2 postfixé à str1.	str1 VARCHAR2, str2 VARCHAR2	
INITCAP	Retourne str1 avec la première lettre de chaque mot en majuscule et toutes les autres en minuscules.	str1 VARCHAR2	
INSTR	Retourne la position de départ de str2 dans str1. La recherche commence en pos pour la nième occurence. Si elle est négative, la recherche est effectuée depuis la fin. À la fois pos et n ont comme valeur par défaut 1. La fonction retourne 0 si str2 n'est pas trouvé.	str1 VARCHAR2, str2 VARCHAR2 [, pos NUMBER [, n NUMBER]]	
INSTRB	Similaire à INSTR sauf que pos est une position exprimée en nombre d'octets.	str1 VARCHAR2, str2 VARCHAR2 [, pos NUMBER [, n NUMBER]]	
LENGTH	Retourne le nombre de caractères dans str et pour le type de données CHAR ; la longueur inclut les blancs de fins.	str CHAR ou str VARCHAR2	NUMBER
LENGTHB	Similaire à LENGTH ; retourne un nombre d'octets pour str, incluant les blancs de fins pour CHAR.	str CHAR ou str VARCHAR2	NUMBER
LOWER	Retourne str avec toutes les lettres en minuscules.	str CHAR ou str VARCHAR2	CHAR ou VARCHAR2
LPAD	str est complété à gauche à la longueur len avec les caractères de pad, qui est par défaut un unique blanc. Retourne les len premiers caractères dans str si str est plus long que len.	str VARCHAR2 len NUMBER [, pad VARCHAR2]	

LTRIM	Retourne `str` avec des caractères retirés jusqu'au premier caractère qui ne se trouve pas dans `set` ; `set` contient par défaut un blanc.	`str VARCHAR2 [, set VARCHAR2]`	
NLS_INITCAP	Similaire à `INITCAP` sauf que la séquence de tri est spécifiée par `nlsparms`.	`str VARCHAR2 [, nlsparms VARCHAR2]`	
NLS_LOWER	Similaire à `LOWER` sauf que la séquence de tri est spécifiée par `nlsparms`.	`str VARCHAR2 [, nlsparms VARCHAR2]`	
NLS_UPPER	Similaire à `UPPER` sauf que la séquence de tri est spécifiée par `nlsparms`.	`str VARCHAR2 [, nlsparms VARCHAR2]`	
NLSSORT	Retourne `str` avec un tri spécifié par `nlsparms`.	`str VARCHAR2 [, nlsparms VARCHAR2]`	`RAW`
REPLACE	Renvoie `str1` où toutes les occurences de `str2` sont remplacées par `str3`. Si `str3` n'est pas spécifié, toutes les occurences de `str2` sont retirées.	`str1 VARCHAR2, str2 VARCHAR2, [str3 VARCHAR2]`	
RPAD	Similaire à `LPAD` sauf que `str` est complété à droite avec `pad` pour atteindre une longueur de `len`.	`str VARCHAR2, len VARCHAR2, [, pad VARCHAR2]`	`NUMBER`
RTRIM	Similaire à `LTRIM` sauf que les caractères de fin sont retirés de `str` après le premier caractère ne se trouvant pas dans `set`.	`str VARCHAR2 [, set VARCHAR2]`	
SOUNDEX	Retourne le code phonétique « soundex » de `str`.	`str VARCHAR2`	
SUBSTR	Retourne une sous-chaîne de `str` commençant à la position `pos` et de longueur `len` ou allant jusqu'à la fin de `str` si `len` est omis. Si `pos` < 0, `SUBSTR` compte en arrière à partir de la fin de `str`.	`str VARCHAR2, pos NUMBER [, len NUMBER]`	
SUBSTRB	Similaire à `SUBSTR` sauf que l'on travaille en octets et non en caractères.	`str VARCHAR2, pos NUMBER [, len NUMBER]`	
TRANSLATE	Remplace toutes les occurences de `set1` avec des caractères de `set2` dans `str`.	`str VARCHAR2, set1 VARCHAR2, set2 CHAR`	

UPPER	Renvoie toutes les lettres en majuscule.	str CHAR ou str VARCHAR2	

13.2.2 Fonctions de conversion

La table qui suit donne la liste des fonctions de conversion disponibles avec une brève description, la liste des arguments et la valeur de retour. Les arguments optionnels sont indiqués entre crochets. Toutes les fonctions de conversion internes sont de la forme suivante :

```
function CHARTOROWID (str VARCHAR2) return ROWID
```

Fonction	Description	Argument(s)	Valeur de retour
CHARTOROWID	Convertit str dans le type ROWID.	str CHAR ou str VARCHAR2	ROWID
CONVERT	Convertit str de l'ensemble de caractères set1 vers l'ensemble de caractères set2. set1 et set2 peuvent être des noms d'ensembles de caractères ou des noms de colonnes.	str VARCHAR2, set1 VARCHAR2, set2 VARCHAR2	VARCHAR2
HEXTORAW	Convertit str de CHAR ou VARCHAR2 en RAW.	str CHAR ou str VARCHAR2	RAW
RAWTOHEX	Opposé de HEXTORAW.	bin RAW	VARCHAR2
ROWIDTOCHAR	Convertit bin de ROWID en chaîne hexadécimale de 18 octets.	bin ROWID	VARCHAR2
TO_CHAR (Dates)	Convertit dte en VARCHAR2 en utilisant fmt. La langue pour la conversion de date peut être spécifiée en nlsparms.	dte DATE [, fmt VARCHAR2 [, nlsparms]]	VARCHAR2
TO_CHAR (Numbers)	Convertit num en VARCHAR2 en utilisant fmt. Les éléments de format suivant peuvent être spécifiés dans nlsparms : caractère décimal, séparateur de groupe et un symbole pour la monnaie locale ou internationale.	num NUMBER [, fmt VARCHAR2 [, nlsparms]]	VARCHAR2
TO_CHAR (Labels)	Convertit le type MLSLABEL en VARCHAR2 en utilisant fmt.	label [, fmt VARCHAR2]	VARCHAR2

TO_DATE	Convertit str ou num en une valeur DATE en utilisant fmt. L'argument fmt n'est pas optionnel en convertissant un nombre. La langue pour la conversion de date peut être spécifiée en nlsparms.	str VARCHAR2 ou num NUMBER [,nlsparms]	DATE
TO_LABEL	Convertit str dans le type de donnée MLSLABEL. Si fmt est omis, str doit être le format de l'étiquette par défaut.	str CHAR ou str VARCHAR2 [, fmt VARCHAR2]	MLSLABEL
TO_MULTI_BYTE	Convertit str d'un codage 8 bits en un codage multi-octets, s'il existe.	str CHAR str VARCHAR2	CHAR VARCHAR2
TO_NUMBER	Convertit str en une valeur numérique en fonction de la valeur de fmt. Les éléments du format peuvent être spécifiés dans nlsparms comme décrit dans la fonction TO_CHAR.	str CHAR str VARCHAR2	NUMBER NUMBER
TO_SINGLE_BYTE	Opposé de TO_MULTI_BYTE.	str CHAR str VARCHAR2	CHAR VARCHAR2

13.2.3 Fonctions de date

Toutes les fonctions de date renvoient une valeur de type DATE sauf en cas de spécification contraire dans la table ci-dessous, qui donne la liste des fonctions de date disponibles avec une brève description, la liste des arguments et la valeur de retour. Les arguments optionnels sont mis entre crochets. Toutes les fonctions de date internes sont de la forme suivante :

```
function ADD_MONTHS (dte DATE, num NUMBER) return DATE
```

Fonction	Description	Argument(s)	Valeur de retour
ADD_MONTHS	Retourne dte plus ou moins num mois.	dte DATE, num NUMBER	
LAST_DAY	Retourne le dernier jour du mois de dte.	dte DATE	
MONTHS_BETWEEN	Retourne le nombre de mois entre dte1 et dte2. NUMBER est < 0 si dte1 est antérieur à dt2.	dte1 DATE, dte2 DATE	NUMBER
NEW_TIME	Retourne la date et l'heure dans la zone zon2 en fonction de la date et heure dte exprimée dans la zone zon1.	dte DATE, zon1 VARCHAR2, zon2 VARCHAR2	

NEXT_DAY	Retourne le premier jour de la semaine pour le jour day qui suit la date dte.	dte DATE, day VARCHAR2	
ROUND	Retourne dte arrondi à l'unité spécifiée dans le format fmt. Si aucun format n'est spécifié, dte est arrondi au jour le plus proche.	dte DATE [, fmt VARCHAR2]	
SYSDATE	Retourne la date et l'heure courante du système.	Pas d'arguments.	
TRUNC	Retourne dte où l'heure du jour est tronquée comme spécifié dans fmt.	dte DATE [, fmt VARCHAR2]	

13.2.4 Fonctions diverses

La table ci-dessous donne la liste des fonctions diverses avec une brève description, une liste d'arguments et une valeur de retour. Les arguments optionnels sont donnés entre crochets.

Fonction	Description	Argument(s)	Valeur de retour
DUMP	Retourne une représentation interne de expr basée sur l'une des spécifications fmt suivantes : 8 = octal, 10 = décimal, 16 = hexadécimal, 17 = caractère simple. Les arguments pos et len spécifient la portion de la représentation à renvoyer.	expr DATE ou expr NUMBER ou expr VARCHAR2 [, fmt BINARY_INTEGER [, pos BINARY_INTEGER [, len BINARY_INTEGER]]]	VARCHAR2
GREATEST	Renvoie la plus grande valeur de exprn. Toutes les expressions doivent être de type compatible avec expr1.	expr1, expr2, expr3, ...	
GREATEST_LB	Retourne la plus grande borne inférieure de la liste d'étiquettes. Chaque étiquette doit être de type MLSLABEL.	label [, label] . . .	MLSLABEL
LEAST	Retourne la plus petite valeur de la liste de exprn. Toutes les expressions doivent être de type compatible avec expr1.	expr1, expr2, expr3 . . .	

LEAST_UB	Retourne la plus petite borne supérieure de la liste des étiquettes. Chaque étiquette doit être de type MLSLABEL.	label [, label] . . .	MLSLABEL
NVL	Retourne la valeur de arg1 si cette valeur n'est pas nulle et arg2 dans le cas contraire. arg1 et arg2 doivent être de même type.	arg1, arg2	Type de données de arg1 et arg2
UID	Renvoie un unique numéro d'identification pour l'utilisateur actuel d'Oracle.	Pas d'arguments	NUMBER
USER	Retourne le nom de l'utilisateur courant d'Oracle.	Pas d'arguments	VARCHAR2
USERENV	Retourne des informations sur la session actuelle basées sur str, qui peut avoir l'une des valeurs suivantes : 'ENTRYID' (identificateur d'entrée), 'LABEL' (étiquette de la session), 'LANGUAGE' (langue, territoire et ensemble des caractères de la base), 'SESSIONID' (identificateur de session), 'TERMINAL' (type de terminal de la session)	str VARCHAR2	VARCHAR2
VSIZE	Retourne le nombre d'octets dans expr.	expr DATE ou expr NUMBER ou expr VARCHAR2	NUMBER

13.2.5 Fonctions numériques

Toutes les fonctions numériques renvoient une valeur de type NUMBER sauf indication contraire dans la table ci-dessous, qui donne la liste des fonctions numériques disponibles avec une brève description, la liste des arguments et la valeur de retour. Les arguments optionnels sont donnés entre crochets. Toutes les fonctions numériques internes sont de la forme suivante :

```
function ABS (n NUMBER) return NUMBER
```

Fonction	Description	Argument(s)
ABS	Retourne la valeur absolue de n.	n NUMBER
CEIL	Retourne le plus petit entier $\geqslant$ n.	n NUMBER
COS	Retourne le cosinus de a. L'angle a doit être en radians.	a NUMBER

COSH	Retourne le cosinus hyperbolique de n.	n NUMBER
EXP	Retourne la valeur de e^n .	n NUMBER
FLOOR	Retourne le plus grand entier $\leqslant$ n.	n NUMBER
LN	Retourne le logarithme népérien de n lorsque $n > 0$.	n NUMBER
LOG	Retourne le logarithme en base m de n lorsque $m > 1$ et $n > 0$.	m NUMBER, n NUMBER
MOD	Retourne le reste de m/n.	m NUMBER, n NUMBER
POWER	Retourne la valeur de m^n .	m NUMBER, n NUMBER
ROUND	Retourne m arrondi à n chiffres.	m NUMBER, n NUMBER
SIGN	Retourne -1 si $n < 0$, 0 si $n = 0$ et 1 si $n > 0$.	n NUMBER
SIN	Retourne le sinus de a. L'angle a doit être en radians.	a NUMBER
SINH	Retourne le sinus hyperbolique de n.	n NUMBER
SQRT	Retourne la racine carrée de n.	n NUMBER
TAN	Retourne la tangente de a. L'angle a doit être en radians.	a NUMBER
TANH	Retourne la tangente hyperbolique de n.	n NUMBER
TRUNC	Retourne m tronqué à n chiffres.	m NUMBER [, n NUMBER]

14 Compléments

14.1 Instruction DECLARE

L'utilisation de l'instruction DECLARE est limitée à la création de sous-blocs au sein de blocs PL/SQL, comme montré dans l'exemple suivant :

```
BEGIN
  ...
  <<inner>>
  DECLARE
    ...

    BEGIN
      ...
    END inner;
  ...
END;
```

Ce code utilise DECLARE pour déclarer des curseurs, des variables et des constantes locales au sous-bloc appelé « inner ».

14.2 Conventions de nommage

PL/SQL permet de référencer tous les objets définis (tels que des variables, des curseurs, des *packages*, etc.) en utilisant de simples références, des références qualifiées, des références distantes ou bien une combinaison de références qualifiées et de références distantes. La capitalisation ne joue de rôle dans aucune référence d'objets. Une simple référence à l'interface du *package* order_total prend la forme suivante :

```
get_order_total( ... );
```

Une référence qualifiée au même *package* se présente ainsi :

```
order_total.get_order_total( ... );
```

Une référence distance à ce *package* est montrée dans l'exemple suivant :

```
get_order_total@concepts( ... );
```

Finalement, en utilisant une référence qualifiée et distante, on référence le *package* `order_total` comme suit :

```
order_total.get_order_total@concepts( ... );
```

Les deux premières instances référencent la procédure `order_total` sur la machine locale. Les deux dernières instances montrent un accès distant à la procédure `order_total` en utilisant le lien vers la base `concepts`.

14.2.1 Synonymes

Pour simplifier encore davantage la référence à une procédure, on peut utiliser un synonyme. Il n'est toutefois pas possible d'utiliser des synonymes pour référencer des objets `PL/SQL` contenus dans des sous-programmes ou *package*. Un exemple de création de synonyme est donné à la section § 2.2 (« création de *packages* ») qui montre un exemple de script pour construire le *package* `order_total`.

14.2.2 Portée d'une référence

Une autre convention de nommage qu'il est utile de mentionner est la portée d'une référence. Ceci fait référence à l'étendue sur laquelle on peut faire référence à un identificateur `PL/SQL` tel qu'une variable ou un sous-programme. En termes simples, la hiérarchie des portées est bloc, local, global et application. L'exemple suivant illustre ce point :

```
CREATE OR REPLACE PACKAGE order_total
AS
  PROCEDURE
    get_order_total ( ... );
END order_total
CREATE OR REPLACE PACKAGE BODY order_total
AS
  ot_failure EXCEPTION;
  ...
  PROCEDURE
    init_line_items
  IS
    i BINARY INTEGER := 0;
      ...
    BEGIN
      ...
      <<inner>>
      DECLARE
        j BINARY_INTEGER := 0;
      BEGIN
        j = i;
        ...
      EXCEPTION
      ...
      raise ot_failure;
```

```
      END inner;
      ...
    END;
END order_total;
```

Dans cet exemple, la portée de référence pour la variable `j` est le sous-bloc interne où la variable est définie. La variable `i`, par contre, est définie locale à la procédure `init_line_items`. Elle peut être référencée dans le sous-bloc interne comme montré. L'exception définie, `ot_failure`, est globale au corps du *package* et peut être référencée par tous les sous-programmes, mais pas par le code appelant. Enfin, la routine d'interface `get_order_total` et les variables associées sont disponibles au niveau de l'application et dans le *package*.

Une dernière note sur la portée de référence : dans un sous-bloc, on peut définir des identificateurs locaux de même noms que des identificateurs globaux. Il faut alors faire référence aux identificateurs globaux avec un nom qualifié qui peut être un bloc englobant ou un nom de sous-programme. La référence qualifiée est alors faite en utilisant la notation « . ».

14.3 Conversion de types de données

`PL/SQL` supporte à la fois des conversions explicites et implicites de types de données. Les conversions explicites se produisent en employant une fonction interne, telle que la fonction `TO_NUMBER` décrite précédemment. Les conversions se produisent au moment de la compilation lorsqu'un type de données est fourni et un autre type de données est attendu. Cette caractéristique de `PL/SQL` permet de se reposer sur le compilateur plutôt que d'utiliser des routines de conversion explicites. Considérons l'instruction `SQL` suivante :

```
SELECT SUM(grand_total) FROM order
WHERE order_date < '10-SEP-95';
```

Dans ce cas, la colonne `order_date` est stockée sous la forme du type `DATE` et elle est comparée à `'10-SEP-95'`, qui est une valeur littérale `CHAR`. `PL/SQL` effectue une conversion implicite de ce littéral vers le type `DATE` lorsque la procédure contenant ce `SELECT` est compilée.

14.4 *Triggers* (déclencheurs) de bases de données

Une autre utilisation courante de procédures `PL/SQL` est la création de *triggers* (déclencheurs) de bases de données. Les *triggers* sont des *packages* qui se déclenchent automatiquement lorsqu'une table d'une base de données satisfait certains critères après une opération `SQL`. Par conséquent, les *triggers* de bases de données ne sont pas référencés explicitement par d'autres procédures ou applications.

Il est possible de lier jusqu'à 12 *triggers* de bases de données à une table de donnée.

Il y a trois parties distinctes à considérer lors de la construction d'un *trigger*. Il y a tout d'abord l'événement qui cause le déclenchement du *trigger*. Le déclenchement conduit à l'exécution de l'action que l'on peut considérer comme le code du *trigger*. Enfin, il faut considérer les contraintes optionnelles que l'on peut souhaiter placer sur le *trigger*. Voyons de plus près comment les *triggers* sont construits.

Tout comme les *packages*, tous les *triggers* suivent une forme de développement standard. Ce qui suit est un exemple de création de *trigger* :

```
CREATE TRIGGER name
  (événement déclenchant le trigger)
  ...
  (contrainte optionnelle du trigger)
BEGIN
  (action du trigger)
END;
```

Comme le montre cet exemple, tous les *triggers* commencent par l'instruction `CREATE TRIGGER` qui est le point d'entrée du *trigger* de nom `name`. L'événement déclencheur du *trigger* commence avec un mot-clé pour spécifier quand le *trigger* doit se déclencher. Cette section du code identifie l'opération `SQL` qui

passe le contrôle au code de l'action SQL. Toute contrainte sur l'opération SQL est identifiée dans la zone de spécification optionnelle du *trigger*.

Si l'on n'est pas propriétaire de la table dans laquelle on crée le *trigger*, il faut avoir les privilèges ALTER ou ALTER ANY TABLE sur la table. Un autre privilège qui est nécessaire est CREATE TRIGGER, quelle que soit la table dans laquelle le *trigger* est créé.

L'exemple suivant illustre la création et l'utilisation d'un *trigger* :

```
CREATE TRIGGER check_order_total
  AFTER UPDATE OF order_total ON order
  FOR EACH ROW
  WHEN (new.status = 'NEW')
BEGIN
  IF :new.order_total = 0 THEN
    INSERT INTO order_log
      values(:new.order);
    UPDATE order SET :new.status = 'ERR';
  END IF;
END;
```

Cet exemple montre que la spécification de l'événement *trigger* commence avec un mot-clé, ici AFTER, qui détermine quand le *trigger* doit être déclenché. L'instruction FOR EACH ROW cause le déclenchement du *trigger* une fois pour chaque ligne au lieu d'une fois par table comme c'est le cas par défaut. Une contrainte du déclenchement est que l'état (status) de la commande mise à jour doit être 'NEW'. L'action du *trigger* consiste à insérer une ligne dans la table order_log et de mettre à jour l'état de la commande à 'ERR'.

Un nom corrélé tel que :new fait référence à des valeurs de colonnes nouvellement mises à jour. On peut aussi référencer d'anciennes valeurs d'une colonne qui change. Comme indiqué, on n'utilise pas le « : » dans le code des contraintes du *trigger*.

Il n'est pas possible d'utiliser des commandes COMMIT, ROLLBACK ou SAVEPOINT dans des *triggers*.

14.5 Compléments sur les exceptions

14.5.1 Relancement d'exceptions

La commande RAISE a dejà été illustrée pour lever des exceptions dans du code, mais RAISE peut aussi être utilisé pour relancer une exception. Considérons l'exemple suivant :

```
CREATE OR REPLACE PACKAGE order_total
AS
  ot_failure EXCEPTION;
  ...
  BEGIN
    ...
    BEGIN
      ...
      if g_recip_counter > max_lines then
        RAISE ot_failure;
      end if;
    EXCEPTION
      when OT_FAILURE then
        ...
      RAISE;
    END;
    ...
    EXCEPTION
      when OT_FAILURE then
```

```
      ...
  END;
END order_total;
```

Dans cet exemple, l'exception est levée dans un sous-bloc avec une définition de *handler* d'exception `ot_failure`. Après le traitement de cette erreur au sein du *handler*, l'exception est à nouveau levée pour un traitement ultérieur dans le bloc de la procédure principale. Ceci est accompli avec un autre *handler* d'exception `ot_failure`.

14.5.2 Poursuite de l'exécution

Après la levée d'une exception dans un sous-bloc `PL/SQL`, il est possible de continuer l'exécution avant de sortir du bloc conducteur. Il faut placer le code exécutable dans le bloc conducteur après le *handler* d'exception. Ce qui suit est une illustration de cette technique :

```
<<outer>>
BEGIN
  ...
  <<inner>>
  BEGIN
    ...
    if g_recip_counter > max_lines then
      raise ot_failure;
    end if;
  EXCEPTION
    when OT_FAILURE then
      ...
  END inner;
  UPDATE order SET status = 'SU';
  INSERT INTO suspense_queue
  VALUES (order,g_out_msg);
EXCEPTION
  ...
END outer;
```

Cet exemple montre que l'exception a été traitée dans le sous-bloc interieur au moment où le contrôle a été passé au bloc conducteur externe. Au moment où le *handler* d'exception interne s'est achevé, l'exécution a repris avec la commande `UPDATE` dans le bloc externe. De cette manière, l'exécution de la procédure peut continuer après la rencontre d'une erreur qui autrement serait fatale.

14.5.3 Réexécution de transactions

Une autre méthode pour poursuivre l'exécution d'une procédure après la levée d'une exception est connue sous le nom de réexécution de transaction. La technique est similaire à la poursuite d'une exécution après la levée d'une exception en ce que la transaction a réessayer doit exister dans le sous-bloc. En utilisant des contrôles de boucles itératifs, on peut répéter une transaction aussi souvent que souhaité après la levée d'une exception. Ce qui suit illustre cette technique :

```
BEGIN
  ...
  FOR i in 1..10 LOOP
    ...
    BEGIN
      SAVEPOINT update_order
        (transactions SQL)
```

```
      COMMIT;
      EXIT;
    EXCEPTION
      WHEN ... THEN
        ROLLBACK to update_order
          (correction des problèmes des données)
      ...
    END;
  END LOOP;
END;
```

Sous le contrôle de la boucle `FOR`, les transactions `SQL` peuvent être essayées au total dix fois avant que l'exécution de la procédure ne soit terminée. `SAVEPOINT update_order` est le point de retour pour des transactions qui ont échoué. Si une erreur est rencontrée durant la phase de transactions `SQL` de ce sous-bloc, le contrôle est transféré au *handler* d'exceptions qui essaie de résoudre les problèmes de données. Après l'exécution du *handler* d'erreur, le contrôle est transféré à la boucle `FOR` pour une autre passe.

14.6 Compléments sur les structures de contrôle

Les structures suivantes sont rarement utilisées mais sont quelquefois nécessaires.

14.6.1 Instruction `EXIT`

L'instruction `EXIT` a déjà été mentionnée comme un moyen de sortir d'une boucle `FOR`. Cette instruction peut aussi être utilisée lors du réessai de transactions après la levée d'exceptions. Dans ce sens, `EXIT` procure un mécanisme pour le transfert inconditionnel du contrôle d'un point du code à un autre.

14.6.2 Contrôle séquentiel

L'utilisation de contrôle séquentiel dans `PL/SQL` n'est pas un élément essentiel au développement de bon code. Toutefois, il est utile de mentionner cette technique dans une description complète du langage. Deux commandes sont disponibles pour le contrôle séquentiel : `GOTO` et `NULL`.

`GOTO` est une commande de branchement inconditionnelle qui transfère le contrôle à une étiquette définie dans la portée de la logique de branchement. L'étiquette doit précéder une commande exécutable ou définir un bloc `PL/SQL` comme illustré dans l'exemple suivant :

```
<<count_lines>>
for i in 1..g_line_counter LOOP
  ...
  if i>max_lines then
    GOTO clean_up;
  end if;
  ...
end LOOP init_lines;
<<clean_up>
g_recip_counter = i-1;
...
```

Dans cet exemple, le branchement conditionnel transfère le contrôle à l'étiquette `clean_up` pour un traitement ultérieur. L'utilisation de la commande `GOTO` n'est pas conseillée car elle peut conduire à du code non structuré. D'autres constructions de `PL/SQL` permettent d'écrire du code qui est plus facile à comprendre et maintenir.

Surtout limité à l'amélioration de la lisibilité du code, la commande `NULL` est un moyen de montrer que tous les choix possibles ont été considérés. `NULL` est considéré comme une commande exécutable.

```
if g_recip_counter > max_lines then
```

```
  g_recip_counter = max_lines;
else
  NULL;
end if;
```

Cet exemple utilise `NULL` pour montrer qu'il n'y a rien à faire si `g_recip_counter` est $\leqslant$ `max_lines`. De manière évidente, ce code peut être achevé sans la clause `ELSE` mais le fait d'utiliser `NULL` montre que d'autres options ont été considérées.