Лекция 4. Введение в PL/SQL.

Институт FORWARD

1 Язык PL/SQL

PL/SQL (Procedural Language / Structured Query Language) — язык программирования, процедурное расширение языка SQL, разработанное корпорацией Oracle.
Язык PL/SQL обладает следующими определяющими характеристиками:

- Высокая структурированность, удобочитаемость и доступность. Новичок сможет легко постигнуть азы своей профессии с PL/SQL — этот язык прост в изучении, а его ключевые слова и структура четко выражают смысл кода. Программист с опытом работы на других языках очень быстро привыкнет к новому синтаксису.
- Стандартный переносимый язык разработки приложений для баз данных Oracle. Если вы написали на PL/SQL процедуру или функцию для базы данных Oracle, находящейся на портативном компьютере, то эту же процедуру можно будет перенести в базу данных на компьютере корпоративной сети и выполнить ее без каких-либо изменений (конечно, при условии совместимости версий Oracle). «Написать один раз и использовать везде» — этот основной принцип PL/SQL был известен задолго до появления языка Java. Впрочем, «везде» в данном случае означает «при работе с любой базой данных Oracle».
- Встроенный язык. PL/SQL не используется как самостоятельный язык программирования. Это встроенный язык, работающий только в конкретной управляющей среде. Таким образом, программы PL/SQL можно запускать из базы данных (скажем, через интерфейс SQL*Plus). Также возможно определение и выполнение программ PL/SQL из формы или отчета Oracle Developer (клиентский PL/SQL). Однако вы не сможете создать исполняемый файл программы на PL/SQL и запускать его автономно.
- Высокопроизводительный, высокоинтегрированный язык баз данных. В настоящее время существует много способов написания программ, работающих с базами данных Oracle. Например, можно использовать Java и JDBC или Visual Basic. Одним из важнейших аспектов PL/SQL является его тесная интеграция с SQL. Для выполнения SQL-инструкций в программах на PL/SQL не требуется никакой промежуточной
  программной «прослойки» вроде ODBC (Open Database Connectivity) или JDBC (Java Database Connectivity)

.PL/SQL содержит полный набор команд, предназначенных для управления последовательностью выполнения строк программы. В него входят следующие команды:

- IF и CASE. Реализация условной логики выполнения — например, «Если количество книг больше 1000, то...»
- Полный набор команд циклов и итеративных вычислений. К этой группе относятся команды FOR, WHILE и LOOP.
- GOTO.

Язык PL/SQL предоставляет разработчикам мощный механизм оповещения о возникающих ошибках и их обработки. При возникновении ошибки — как системной, так и ошибки в приложении — в PL/SQL инициируется исключение. В результате выполнение блока прерывается, и управление передается для обработки в раздел исключений текущего блока, если он имеется. Блок обработки исключений начинается со слова EXCEPTION. В этом блоке указывается идентификатор ошибки и действия, которые будут выполнены при возникновении этой ошибки. Для генерации исключения используется процедура raise_application_error или raise.
Прежде чем подробнее разобрать особенности языка, напомним синтаксис DML операции.

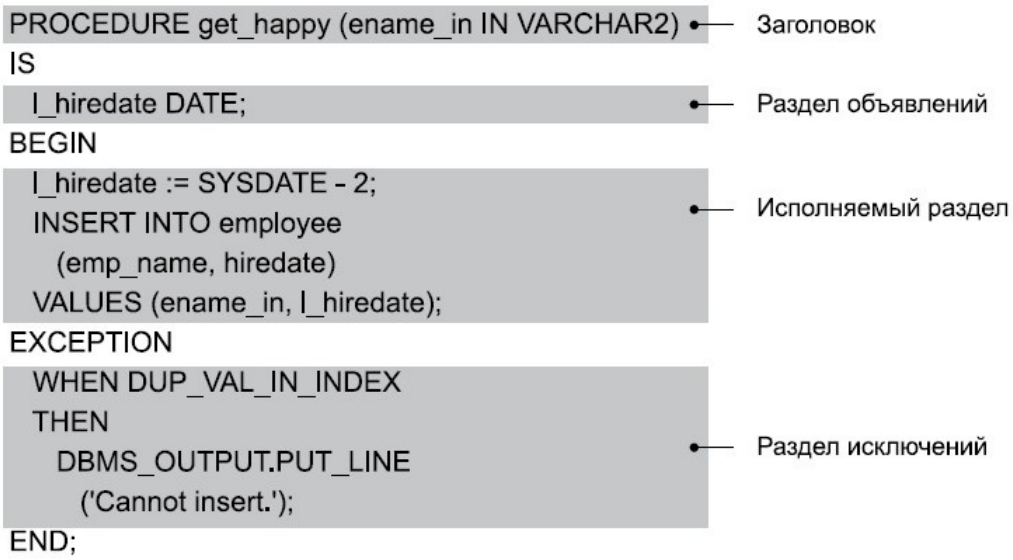
Рассмотрим подробнее особенности языка.

2 Структура блока PL/SQL

В PL/SQL, как и в большинстве других процедурных языков, наименьшей единицей группировки кода является блок. Он представляет собой фрагмент программного кода, определяющий границы выполнения и области видимости для объявлений переменных и об- работки исключений. PL/SQL позволяет создавать как именованные, так и анонимные блоки (то есть блоки, не имеющие имени), которые представляют собой пакеты, процедуры, функции, триггеры или объектные типы.
Блок PL/SQL может содержать до четырех разделов, однако только один из них является обязательным.

- Заголовок. Используется только в именованных блоках, определяет способ вызова именованного блока или программы. Не обязателен.
- Раздел объявлений. Содержит описания переменных, курсоров и вложенных блоков, на которые имеются ссылки в исполняемом разделе и разделе исключений. Необязателен.
- Исполняемый раздел. Команды, выполняемые ядром PL/SQL во время работы приложения. Обязателен.
- Раздел исключений. Обрабатывает исключения (предупреждения и ошибки). Необязателен.

Таким образом, в любой исполняемый блок, в том числе из одной строчки, мы можем добавить раздел объявлений или исключений.



3 Анонимные блоки

В анонимном блоке PL/SQL, нет раздела заголовка, блок начинается ключевым словом DECLARE (или BEGIN). Анонимный блок не может быть вызван из другого блока, поскольку он не имеет идентификатора, по которому к нему можно было бы обратиться. Таким образом, анонимный блок представляет собой контейнер для хранения команд PL/SQL — обычно с вызовами процедур и функций.
Анонимные блоки используются для:

Передачи заказчику скриптов. В случае, если необходимо провести какую-то операцию с данными единоразово, именованный блок(процедуру или функцию) делать смысла не имеет. В таком случае, заказчику передается скрипт в виде анонимного блока. Например, следующий скрипт изменяет в названии услуг слово "BOX"на "Приставка".

```
BEGIN
  -- Обращение N-432
  UPDATE fw_service
     SET v_name = REPLACE(UPPER(v_name), 'BOX', 'Приставка')
   WHERE UPPER(v_name) LIKE '%BOX%';
END;
```

Поскольку анонимные блоки могут содержать собственные разделы объявлений, разработчики часто используют вложение анонимных блоков для ограничения области видимости идентификаторов. При этом, переменная, объявленная в блоке видна во всех вложенных блоках, но, как правило, не видна для внешних. В приведенном ниже примере объявленная во внутреннем блоке переменная C (т.е. переменная с таким же именем) перекрывает видимость переменной C, определенной во внешнем блоке. В точке 3 видимы переменные A и C типа date, которая снова становится видна, т.к. закончилась область действия переменной C типа char(2).

```
DECLARE
  -- начало внешнего блока
  A number;
  C date;
BEGIN
  -- 1
  DECLARE
    -- начало внутреннего блока
    B number;
    C char(2);
  BEGIN -- 2
  END; -- конец внутреннего блока
  -- 3
END; -- конец внешнего блока
```

Поскольку анонимные блоки могут содержать собственные разделы исключений, разработчики часто используют вложение анонимных блоков для организации обработки исключений в более крупных программах. Это, пожалуй, самое частое применение анонимного блока. Не стоит навешивать общий обработчик ошибок на всю вашу программу. Редактирующему ваш код, в таком случае, не будет понятно, к чему относится какое исключение. Поэтому необходимо обрабатывать исключения именно в тех блоках кода, в которых оно может возникнуть. Прочитать подробнее о исключениях можно по ссылке в разделе "Материалы" в личном кабинете стажера.

Рассмотрим примеры анонимных блоков:

```
Простейший анонимный блок:
BEGIN
    DBMS_OUTPUT.PUT_LINE(SYSDATE);
END;

Анонимный блок с добавлением раздела объявлений:
DECLARE
  l_right_now VARCHAR2(9);
BEGIN
  l_right_now := SYSDATE;
  DBMS_OUTPUT.PUT_LINE(l_right_now);
END;
Тот же блок, но с разделом исключений: DECLARE
DECLARE
  l_right_now VARCHAR2(9);
BEGIN
  l_right_now := SYSDATE;
  DBMS_OUTPUT.PUT_LINE(l_right_now);
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('l_right_now не хватает места ' ||
                         ' для стандартного формата даты');
END;
```

4 Именованные блоки

Процедуры, функции, пакеты и другие объекты БД являются именованными блока- ми, хранящимися в базе данных. Их часто называют единым термином – хранимые подпрограммы. Они являются объектами базы данных и, как и все объекты БД, имеют имена. Хранимые процедуры могут иметь входные и выходные аргументы и могут вызываться из других программ. Как и все другие объекты БД, процедуры, функции и пакеты создаются командой CREATE, а удаляются командой DROP. Пакеты мы рассмотрим в следующей лекции, сейчас подробнее остановимся на процедурах и функциях. Синтаксис команды создания процедур следующий:

```
CREATE [OR REPLACE] PROCEDURE имя_процедуры
[ (параметр [{IN | OUT | IN OUT}] тип [DEFAULT значение],
…
параметр [{IN | OUT | IN OUT}] тип [DEFAULT значение])] {IS | AS}
/* раздел объявлений. Может быть пустым */
BEGIN
/* выполняемы раздел */
EXCEPTION
/* раздел исключений. Необязателен */
END [имя_процедуры];
```

Если сравнить структуру безымянного блока и структуру команды CREATE PROCEDURE, то можно отметить их сходство. Отличием является то, что вместо ключевого слова DECLARE используется набор ключевых слов команды CREATE, и после послед- него из этого набора ключевого слова IS (или AS) начинается раздел объявлений. Однако, собственно, команда CREATE – непростая конструкция. Команда может со- держать необязательную фразу OR REPLACE, Если эта фраза указана, то процедура создается, даже если в БД уже есть процедура с таким же именем. В этом случае вновь создаваемая процедура заместит ту, которая уже есть в БД. Если же фраза OR REPLACE отсутствует, то сначала нужно удалить из БД процедуру с таким же именем командой DROP, а затем выполнять команду CREATE PROCEDURE. Имя создаваемой процедуры должно быть уникальным среди объектов базы данных. После ключевого слова END имя процедуры может быть указано, что на практике удобно, если процедура очень длинная. Создаваемая процедура может иметь параметры, которые определяются так же, как и переменные в разделе объявлений – имя и тип данных. Но есть два отличия. Первое – это то, что при указании типа данных NUMBER, CHAR или VARCHAR2 не указывается их размер. Второе отличие заключается в указании, является ли параметр входным, выходным или и тем и другим. Рассмотрим подробнее входные и выходные аргументы:

- **IN.**

Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается в качестве константы PL/SQL – параметра только для чтения – и не может быть изменен. Когда процедура завершается и управление программой возвращается в вызывающую среду, фактический параметр не изменяется. В нашей компании принято давать in параметрам префикс p. Например, pid_contract_inst.

- **OUT.**

Любое значение, которое имеет фактический параметр при вызове процеду- ры, игнорируется. Внутри процедуры формальный параметр рассматривается как неинициированная переменная PL/SQL, то есть содержит NULL-значение, и можно как записать в него значение, так и считать значение из него. Когда процедура завершается и управление программой возвращается в вызы- вающую среду, содержимое формального параметра присваивается фактическому параметру. В нашей компании принято давать префикс o. Например, oid_contract_inst

- **IN OUT.**

Этот вид представляет комбинацию видов IN и OUT. Значение фактического параметра передается в процедуру при ее вызове. Внутри процедуры формальный параметр рассматривается в качестве инициализированной переменной, и можно как записать в него значение, так и считать значение из него. Когда процедура завершается и управление программой возвращается в вызывающую среду, содержимое формального параметра присваивается фактическому пара- метру.

Рассмотрим пример процедуры, которая принимает на вход дату date_in и день недели, а возвращает дату следующего what_day с даты date_in в виде out параметра oresult_date.

```
create or replace procedure get_first_day_proc(pdate_in     in DATE,
                                               pwhat_day    in varchar2,
                                               oresult_date out DATE) is
  name_in_use exception;
  pragma exception_init(name_in_use, -01846);
begin
  begin
    oresult_date := NEXT_DAY(pdate_in, pwhat_day);
  exception
    when name_in_use then
      raise_application_error(-20000,
                              'Введен некорректный день недели!');
    when others then
      raise;
  end;
end;
```

Функции очень похожи на процедуры. Функции и процедуры представляют собой различные формы блоков PL/SQL. Однако вызов процедуры сам по себе является оператором PL/SQL, в то время как вызов функции – это часть некоторого выражения. Синтаксис, применяемый при создании хранимой функции, очень похож на синтаксис создания процедуры:

```
CREATE [OR REPLACE] FUNCTION имя_функции
[ (параметр [{IN | OUT | IN OUT}] тип [DEFAULT значение],
…
параметр [{IN | OUT | IN OUT}] тип [DEFAULT значение])]
RETURN возвращаемый_тип {IS | AS}
/* раздел объявлений */
BEGIN
/* выполняемы раздел */
EXCEPTION
/* раздел исключительных ситуаций */
END [имя_функции];
```

Именно потому, что функция используется как часть выражения, в команде CREATE FUNCTION присутствует фраза RETURN возвращаемый_тип. Этой фразой задается тип возвращаемого функцией значения. Внутри тела функции оператор RETURN применяется для возврата управления в вызывающую среду с некоторым значением. Общий синтаксис оператора выглядит следующим образом: RETURN выражение; где выражение – это возвращаемое значение. В функции может быть несколько операторов RETURN, хотя выполняться будет только один из них. Завершение функции без оператора RETURN является ошибкой. Во всем остальном команда CREATE FUNCTION аналогична команде CREATE PROCEDURE. Рассмотрим пример функции, которая также возвращает дату следующего what_day с даты date_in.

```
create or replace function get_first_day_func(pdate_in      in DATE
                                                default current_timestamp,
                                                pwhat_day     in varchar2
                                                default 'Понедельник')
  return DATE is
  result_date DATE;
begin
  begin
    result_date := NEXT_DAY(pdate_in, pwhat_day);
  exception
    when others then
      IF SQLCODE = -01846 THEN
        raise_application_error(-20000,
                                'Введен некорректный день недели!');
      else
        raise;
      end if;
  end;
  return result_date;
end;
```

Стоит ещё раз отметить, что функции необходимо использовать в выражениях, а процедуры использоваться в них не могут. Приведем пример вызова нашей процедуры и функции и вывода в output возвращенных значений:

```
declare
res_date DATE;
begin
  get_first_day_proc(pdate_in => to_date('2019-01-01', 'yyyy-mm-dd'),
  pwhat_day => 'Четверг', oresult_date => res_date);
  dbms_output.put_line(to_char(res_date, 'dd month yyyy'));
  res_date := get_first_day_func(pdate_in => to_date('2019-01-01', 'yyyy-
mm-dd'),
  pwhat_day => 'Четверг');
  dbms_output.put_line(to_char(res_date, 'dd month yyyy'));
  --а теперь вызовем в самом выражении, при этом с default параметрами
  -- и получим следующий понедельник
  dbms_output.put_line(to_char(get_first_day_func, 'dd month yyyy'));
end;
```

При этом, в переменной res_date будет лежать одно и то же значение.

Кроме того, функции могут быть использованы в запросах:

```
select get_first_day_func(to_date('2016-01-01', 'yyyy-mm-dd')
                          + (level - 1) * 7, 'Четверг')
from dual connect by level <= 10
```

Кроме того, в данном запросе мы можем увидеть, что не обязательно(но желательно) передавать в функцию подстановки pdate_in => и pwhat_day => явно. Мы можем просто перечислить аргументы через запятую.

4.1 Когда использовать функцию?

- Если вы собираетесь использовать ее в sql
- В случае, если вы собираетесь использовать ее в выражениях в pl sql

- В случае, если вы возвращаете одно значение(в том числе составного типа). Это обусловлено тем, что при просмотре кода, сразу видно, какое значение вернула функция. В случае процедуры же необходимо просмотреть ее аргументы, чтобы понять, какие из них она изменила.

4.2 Когда использовать процедуру?

- Если необходимо изменить несколько параметров(в том числе составного типа)
- В случае, если изменять параметры не нужно(все аргументы IN)
- Для сообщения с внешними системами, как правило, используются процедуры

5 Имена атрибутов и переменных.

Переменным и атрибутам стоит давать названия, которые характеризуют их назначение. Например, переменную, которая будет хранить количество чего-либо можно назвать cnt, но не pid_service_inst или безликое x. Кроме того, желательно давать им правильные префиксы.
Но главное - это не называть переменную также, как столбец таблицы, присутству- ющей в коде. Например, рассмотрим следующую процедуру:

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
DELETE orders WHERE order_id = order_id; -- Катастрофа!
END;
```

Этот фрагмент удалит из таблицы orders все записи независимо от переданного значения order_id.
Дело в том, что механизм разрешения имен SQL сначала проверяет имена столбцов и только потом переходит к идентификаторам PL/SQL. Условие WHERE (order_id = order_id) всегда истинно, поэтому все данные пропадают. Возможное решение проблемы выглядит так:

```
PROCEDURE remove_order (order_id IN NUMBER)
IS
BEGIN
DELETE orders WHERE order_id = remove_order.order_id;
END;
```

Но ещё лучше, поскольку атрибут входной, дать ей префикс:

```
PROCEDURE remove_order (porder_id IN NUMBER)
IS
BEGIN
DELETE orders WHERE order_id = porder_id;
END;
```

Для переменных, объявленных внутри процедуры, можно использовать другие префиксы, например, vorder_id или l_order_id

6 Вложенные блоки.

PL/SQL, как и языки Ada и Pascal, относится к категории языков с блочной структурой, то есть блоки PL/SQL могут вкладываться в другие блоки. Стоит упомянуть о такой полезной возможности PL/SQL, как вложенные программы (nested programs). Вложенная программа представляет собой процедуру или функцию, которая полностью размещается в разделе объявлений внешнего блока. Вложенная программа может обращаться ко всем переменным и параметрам, объявленным ранее во внешнем блоке.
Так пакет состоит из функций и процедур, каждая из которых состоит из анонимных блоков(а иногда и именованных тоже).

```
DECLARE
   CURSOR emp_cur IS ...;
BEGIN
   DECLARE
      total_sales NUMBER;
   BEGIN
      DECLARE
         l_hiredate DATE;
      BEGIN
         ...
      END;
   END;
END;
```

В качестве примера навесим собственный обработчик ошибок на вызов функции и процедуры в нашем предыдущем анонимном блоке. Кроме того, в середине про- граммы мы меняем тип res_date на number, что вызывает исключение при попытке преобразовать его в to_char. Но затем мы выходим из области видимости, в которой res_date - число, и он снова становится датой:

```
declare
  res_date DATE;
begin
  begin
    get_first_day_proc(pdate_in     => to_date('2019-01-01', 'yyyy-mm-
dd'),
                       pwhat_day    => 'Четверг',
                       oresult_date => res_date);
  exception
    when others then
      raise_application_error(-20000,
      'Произошла неизвестная ошибка в процедуре get_first_day_proc!');
  end;
  dbms_output.put_line(to_char(res_date, 'dd month yyyy'));
  declare
    res_date number := 1;
  begin
    dbms_output.put_line('res_date = ' || res_date);
    begin
      dbms_output.put_line(to_char(res_date, 'dd month yyyy'));
    exception
      when others then
        dbms_output.put_line(sqlerrm);
    end;
  end;
  dbms_output.put_line('res_date = ' || res_date);
  begin
    res_date := get_first_day_func(pdate_in  => to_date('2019-01-01',
                                                        'yyyy-mm-dd'),
                                   pwhat_day => 'Четверг');
  exception
    when others then
      raise_application_error(-20000,
     'Произошла неизвестная ошибка в функции get_first_day_func!');
  end;
  dbms_output.put_line(to_char(res_date, 'dd month yyyy'));
end;
```

7 Вложенные программы

Вложенная программа представляет собой процедуру или функцию, которая полностью размещается в разделе объявлений внешнего блока. Вложенная программа может обращаться ко всем переменным и параметрам, объявленным ранее во внешнем блоке, как показывает следующий пример:

```
declare
  default_date_in date := current_timestamp;
  function get_first_day_func(pdate_in   in DATE default default_date_in,
                              pwhat_day in varchar2 default 'Понедельник')
    return DATE is
    result_date DATE;
  begin
    begin
      result_date := NEXT_DAY(pdate_in, pwhat_day);
    exception
      when others then
        IF SQLCODE = -01846 THEN
          raise_application_error(-20000,
                                  'Введен некорректный день недели!');
        else
          raise;
        end if;
    end;
    return result_date;
  end;
begin
  dbms_output.put_line(to_char(get_first_day_func(pwhat_day => 'Четверг'),
                               'dd month yyyy'));

end;
```

Вложенные программы упрощают чтение и сопровождение кода, а также позволяют повторно использовать логику, задействованную в нескольких местах блока.
Стоит обратить внимание, что вложенные программы должны быть объявлены после переменных. Присутствовать они могут, как в анонимных, так и в именованных блоках.

8 Символы языка

Программа PL/SQL представляет собой последовательность команд, состоящих из одной или нескольких строк текста. Набор символов, из которых составляются эти строки, зависит от используемого в базе данных набора символов. Каждое ключевое слово, оператор и лексема PL/SQL состоит из разных комбинаций символов данного набора. Запомните, что язык PL/SQL не учитывает регистр символов. Это означает, что символы верхнего регистра интерпретируются так же, как символы нижнего регистра (кроме символов, заключенных в ограничители, с которыми они интерпретируются как литеральная строка). Некоторые символы — как по отдельности, так и в сочетании с другими символами — имеют в PL/SQL специальное значение. Познакомимся с этими символами.

9 Идентификаторы

Идентификатор — это имя объекта данных PL/SQL, которым может быть:

Символы	Описание
;	Завершает объявления и команды
%	Индикатор атрибута (атрибут курсора, подобный %ISOPEN, или атрибут неявных объявлений, например %ROWTYPE) также используется в качестве символа подстановки в условии LIKE
_	Обозначение подстановки одного символа в условии LIKE
@	Признак удаленного местоположения
:	Признак хост-переменной, например :block.item в Oracle Forms
**	Оператор возведения в степень
<> или != или ^= или ~=	Оператор сравнения «не равно»
||	Оператор конкатенации
<< и >>	Ограничители метки
<= и >=	Операторы сравнения «меньше или равно» и «больше или равно»
:=	Оператор присваивания
=>	Оператор ассоциации
..	Оператор диапазона
--	Признак однострочного комментария
/* и */	Начальный и конечный ограничители многострочного комментария

-
    -
        - константа или переменная;
        - исключение;
        - курсор;
        - имя программы: процедура, функция, пакет, объектный тип, триггер и т. д.;
        - зарезервированное слово;
        - метка.

Идентификаторы PL/SQL обладают следующими свойствами:

-
    -
        - длина — до 30 символов;
        - должны начинаться с буквы;
        - могут включать символы «$», «_» и «#»;
        - не должны содержать пропусков.

Если два идентификатора различаются только регистром одного или нескольких символов, PL/SQL обычно воспринимает их как один идентификатор.
Примеры неправильных идентификаторов:

-
    -
        - 1st_year – Не начинается с буквы
        - procedure-name Содержит недопустимый символ ""
        - minimum_%_due – Содержит недопустимый символ "%"
        - maximum_value_exploded_for_detail – Имя слишком длинное
        - company ID – Имя не может содержать пробелов

Тип	Название типа	Примеры и комментарии
Строки фиксированной длины	CHAR, NCHAR	CHAR(4)
Строки переменной длины	VARCHAR2, NVARCHAR2, CLOB	VARCHAR2(200) – строка, длина которой не превышает 200 символов
Числовые типы данных	NUMBER, PLS_INTEGER	NUMBER(20, 2), NUMBER(10), NUMBER, PLS_INTEGER
Дата	DATE	
Время	TIMESTAMP	
Интервалы	INTERVAL	
Логический	BOOLEAN	
Двоичные данные	RAW, BLOB, BFILE	
Ссылки на курсоры	SYS_REFCURSOR	
XML	XMLType и другие	
URI и URL	URIType и другие	
Универсальный тип	AnyType, AnyData и AnyDataSet	

Типы данных PL/SQL

10 Типы данных.

При объявлении переменной или константы необходимо задать ее тип данных. В PL/SQL определен широкий набор скалярных и составных типов данных; кроме того, вы можете создавать пользовательские типы данных. Многие типы данных PL/SQL (например, BOOLEAN и NATURAL) не поддерживаются столбцами баз данных, но в коде PL/SQL эти типы весьма полезны. PL/SQL поддерживает строки фиксированной и переменной длины, состоящие как из традиционных символов, так и из символов Юникода. К строкам первого типа относятся строки CHAR и NCHAR, а к строкам второго вида — VARCHAR2 и NVARCHAR2.
В PL/SQL поддерживаются как вещественные, так и целочисленные типы данных. Тип NUMBER давно был основным типом для работы с числовыми данными; он может применяться для работы с целыми и вещественными данными как с фиксированной, так и с плавающей запятой.
PL/SQL поддерживает тип данных BOOLEAN. Переменные этого типа могут принимать одно из трех значений (TRUE, FALSE или NULL).
Тип данных SYS_REFCURSOR позволяет объявлять курсорные переменные, кото- рые могут использоваться со статическими и динамическими SQL-командами для улучшения гибкости программного кода.
В таблице выше приведены лишь некоторые из типов данных Oracle, с которыми чаще всего требуется работать на практике и/или которые могут встретиться в legacy- коде.

Выше представлен неполный перечень типов данных PL/SQL. Но даже полный перечень можно было бы расширить за счёт пользовательских типов (фактически тут речь о редуцированной поддержке ООП).
Пример создания собственных типов:

```
subtype default_str is varchar2(255);
create or replace type T_EXCHANGE_INTEGER_TABLE as table of number(10);
```

Более подробно типы данных будут рассмотрены в четвертой лекции.

11 Объявление переменных.

Когда вы объявляете переменную, PL/SQL выделяет память для хранения ее значения и присваивает выделенной области памяти имя, по которому это значение можно извлекать и изменять. В объявлении также задается тип данных переменной; он используется для проверки присваиваемых ей значений. Базовый синтаксис объявления переменной или константы: имя тип_данных \[NOT NULL\] \[:= | DEFAULT значение_по_умолчанию\]; Здесь имя — имя переменной или константы, тип_данных — тип или подтип данных, определяющий, какие значе- ния могут присваиваться переменной. При желании можно включить в объявление выражение NOT NULL; если такой переменной не присвоено значение, то база дан- ных инициирует исключение. Секция значение_по_умолчанию инициализирует переменную начальным значением; ее присутствие обязательно только при объявлении констант. Если переменная объявляется с условием NOT NULL, то при объявлении ей должно быть присвоено начальное значение. Примеры объявления переменных разных типов:

```
DECLARE
-- Простое объявление числовой переменной
l_total_count NUMBER;
-- Число, округляемое до двух разрядов в дробной части:
l_dollar_amount NUMBER (10,2);
-- Дата/время, инициализируемая текущим значением системных часов
-- сервера базы данных. Не может принимать значение NULL
l_right_now DATE NOT NULL DEFAULT SYSDATE;
-- Задание значения по умолчанию с помощью оператора присваивания
l_favorite_flavor VARCHAR2(100) := 'Вы любите кофе?';
-- Переменная пользовательского типа объявляется за два этапа.
-- Сначала тип:
TYPE list_of_books_t IS TABLE OF varchar2%ROWTYPE INDEX BY PLS_INTEGER;
-- А затем конкретный список, с которым мы будем работать в блоке:
books list_of_books_t;
```

Стоит заметить, что обычно при объявлении переменных используется :=, в то время как при простановке значения по умолчанию для атрибутов процедуры и функции используется default(использовать := невозможно)

В Oracle также существует другой метод объявления переменных, называемый объявлением с привязкой (anchored declaration). Он особенно удобен в тех случаях, ко- гда значение переменной присваивается из другого источника данных, например из строки таблицы. Объявляя «привязанную» переменную, вы устанавливаете ее тип данных на основании типа уже определенной структуры данных. Таковой может являться другая переменная PL/SQL, заранее определенный тип или подтип (TYPE или SUBTYPE), таблица базы данных либо конкретный столбец таблицы. В PL/SQL существует два вида привязки:

- Скалярная привязка. С помощью атрибута %TYPE переменная определяется на основании типа столбца таблицы или другой скалярной переменной PL/SQL.
- Привязка к записи. Используя атрибут %ROWTYPE, можно определить переменную на основе таблицы или заранее определенного явного курсора PL/SQL.

Синтаксис объявления переменной с привязкой:

```
|имя_переменной тип_атрибута %TYPE[необязательное_значение_по_умолчанию];|
\\
|имя_переменной имя_таблицы/имя_курсора
%ROWTYPE[необязательное_значение_по_умолчанию];|
```

Преимущества объявления с привязкой:

1. Синхронизация со столбцами базы данных. Переменная PL/SQL часто «представляет» информацию из таблицы базы данных. Если явно объявить перемен- ную, а затем изменить структуру таблицы, это может привести к нарушению работы программы.
2. Нормализация локальных переменных. Допустим, переменная PL/SQL хранит вычисляемые значения, которые используются в разных местах приложения. К каким последствиям может привести повторение (жесткое кодирование) одних и тех же типов данных и ограничений во всех объявлениях?

12 Условный оператор IF

Команда IF реализует логику условного выполнения команд программы. С ее помощью можно реализовать конструкции следующего вида: если условие, то выражение. Команда IF существует в трех формах, представленных в следующей таблице:

Разновидность IF	Характеристики
`IF THEN END IF;`	Простейшая форма команды IF. Условие между IF и THEN определяет, должна ли выполняться группа команд, находящаяся между THEN и END IF. Если результат проверки условия равен FALSE или NULL, то код не выполняется
`IF THEN ELSE END IF;`	Реализация логики «или-или». В зависимости от условия между ключевыми словами IF и THEN выполняется либо код, находящийся между THEN и ELSE, либо код между ELSE и END IF. В любом случае выполняется только одна из двух групп исполняемых команд
`IF THEN ELSIF ELSE END IF;`	Последняя, и самая сложная, форма IF выбирает действие из набора взаимоисключающих условий и выполняет соответствующую группу исполняемых команд. Если вы пишете подобную конструкцию IF в версии Oracle9i Release 1 и выше, подумайте, не заменить ли ее командой выбора CASE

Общий синтаксис конструкции IF-THEN выглядит так:

```
IF условие
THEN
... последовательность исполняемых команд ...
END IF;
```

Здесь условие — это логическая переменная, константа или логическое выражение с результатом TRUE, FALSE или NULL. Исполняемые команды между ключевыми словами THEN и END IF выполняются, если результат проверки условия равен TRUE, и не выполняются — если он равен FALSE или NULL. У правила, согласно которому NULL в логическом выражении дает результат NULL, имеются исключения. Некоторые операторы и функции специально реализованы так, чтобы при работе с NULL они давали результаты TRUE и FALSE (но не NULL). Например, для проверки значения NULL можно воспользоваться конструкцией IS NULL:

```
IF salary > 40000 OR salary IS NULL THEN
    give_bonus(employee_id, 500);
END IF;
```

В этом примере условие salary IS NULL дает результат TRUE, если salary не содержит значения, и результат FALSE во всех остальных случаях. Конструкция IF-THEN-ELSE применяется при выборе одного из двух взаимоисключающих действий. Формат этой версии команды IF:

```
IF условие
THEN
... последовательность команд для результата TRUE ...
ELSE
... последовательность команд для результата FALSE/NULL ...
END IF;
```

Здесь условие — это логическая переменная, константа или логическое выражение. Если его значение равно TRUE, то выполняются команды, расположенные между ключевыми словами THEN и ELSE, а если FALSE или NULL — команды между ключевыми словами ELSE и END IF. Важно помнить, что в конструкции IF-THEN- ELSE всегда выполняется

одна из двух возможных последовательностей команд. После выполнения соответствующей последовательности управление передается команде, которая расположена сразу после ключевых слов END IF. Конструкция IF-THEN-ELSIF удобна для реализации логики с несколькими альтернативными действиями в одной команде IF. Как правило, ELSIF используется с взаимоисключающими альтернативами (то есть при выполнении команды IF истинным может быть только одно из условий). Обобщенный синтаксис этой формы IF выглядит так:

```
IF условие-1
THEN
команды-1
...
ELSIF условие-2
THEN
команды-2
[ELSE
команды_else]
END IF;
```

В каждой секции ELSIF (кроме секции ELSE) за условием должно следовать ключе- вое слово THEN. Секция ELSE в IF-ELSIF означает «если не выполняется ни одно из условий», то есть когда ни одно из условий не равно TRUE, выполняются команды, следующие за ELSE. Следует помнить, что секция ELSE не является обязательной — конструкция IFELSIF может состоять только из секций IF и ELSIF. Если ни одно из условий не равно TRUE, то никакие команды блока IF не выполняются.

```
IF salary BETWEEN 10000 AND 20000 THEN
    give_bonus(employee_id, 1500);
  ELSIF salary BETWEEN 20000 AND 40000 THEN
    give_bonus(employee_id, 1000);
  ELSIF salary > 40000 THEN
    give_bonus(employee_id, 500);
  ELSE
    give_bonus(employee_id, 0);
  END IF;
```

Команды IF можно вкладывать друг в друга.

```
procedure bonus(employee_id in number) is
  enough_money  boolean := check_money(employee_id);
  check_account boolean := check_account_func(employee_id);
begin
  IF enough_money /* булева переменная */
   THEN
    if check_account then
      give_bonus(employee_id, 500);
    else
      send_sorry(employee_id);
    end if;
  END IF;
end;
```

Ключевые слова IF, THEN и END IF не обязательно размещать в отдельных строках. В командах IF разрывы строк не важны, поэтому можно писать и так:

```
IF salary > 40000 THEN give_bonus (employee_id,500); END IF;
```

Размещение всей команды в одной строке отлично подходит для простых конструкций IF — таких, как в приведенном примере. Но любая хоть сколько-нибудь сложная команда гораздо лучше читается, когда каждое ключевое слово размещается в отдельной строке.

13 Команды и выражения CASE

Команда CASE позволяет выбрать для выполнения одну из нескольких последовательностей команд. Эта конструкция присутствует в стандарте SQL с 1992 года, хотя в Oracle SQL она не поддерживалась вплоть до версии Oracle8i, а в PL/SQL — до версии Oracle9i Release 1. Начиная с этой версии, поддерживаются следующие разновидности команд CASE:
– Простая команда CASE — связывает одну или несколько последовательностей команд PL/SQL с соответствующими значениями (выполняемая последовательность выбирается с учетом результата вычисления выражения, возвращающего одно из значений).
60 Глава 3. PL/SQL
– Поисковая команда CASE — выбирает для выполнения одну или несколько последовательностей команд в зависимости от результатов проверки списка логических значений. Выполняется последовательность команд, связанная с первым условием, результат проверки которого оказался равным TRUE.
Простая команда CASE позволяет выбрать для выполнения одну из нескольких последовательностей команд PL/SQL в зависимости от результата вычисления выражения. Приведем пример простой команды CASE, в котором премия начисляется в зависимости от значения переменной employee_type:

```
CASE employee_type
    WHEN 'S' THEN
      award_salary_bonus(employee_id);
    WHEN 'H' THEN
      award_hourly_bonus(employee_id);
    WHEN 'C' THEN
      award_commissioned_bonus(employee_id);
    ELSE
      RAISE invalid_employee_type;
  END CASE;
```

В этом примере присутствует явно заданная секция ELSE, однако в общем случае она не обязательна. отличие данной команды от IF в том, что когда в команде IF от- сутствует ключевое слово ELSE, то при невыполнении условия не происходит ничего, тогда как в команде CASE аналогичная ситуация приводит к ошибке.
Поисковая команда CASE проверяет список логических выражений; обнаружив выражение, равное TRUE, выполняет последовательность связанных с ним команд. В сущности, поисковая команда CASE является аналогом команды CASE TRUE, пример которой приведен в предыдущем разделе. Поисковая команда CASE имеет следующую форму записи:

```
CASE
WHEN выражение_1 THEN
команды_1
WHEN выражение_2 THEN
команда_2
...
ELSE
команды_else
END CASE;
```

Она идеально подходит для реализации логики начисления премии:

```
CASE
    WHEN salary >= 10000 AND salary <= 20000 THEN
      give_bonus(employee_id, 1500);
    WHEN salary > 20000 AND salary <= 40000 THEN
      give_bonus(employee_id, 1000);
    WHEN salary > 40000 THEN
      give_bonus(employee_id, 500);
    ELSE
      give_bonus(employee_id, 0);
END CASE;
```

Поисковая команда CASE, как и простая команда, подчиняется следующим правилам:

- Выполнение команды заканчивается сразу же после выполнения последователь- ности исполняемых команд, связанных с истинным выражением. Если истин- ными оказываются несколько выражений, то выполняются команды, связанные с первым из них.
- Ключевое слово ELSE не обязательно. Если оно не задано и ни одно из выра- жений не равно TRUE, инициируется исключение CASE_NOT_FOUND.
- Условия WHEN проверяются в строго определенном порядке, от начала к концу.

Выражения CASE решают ту же задачу, что и команды CASE, но только не для исполняемых команд, а для выражений. Простое выражение CASE выбирает для вычисления одно из нескольких выражений на основании заданного скалярного значения. Поисковое выражение CASE последовательно вычисляет выражения из спис- ка, пока одно из них не окажется равным TRUE, а затем возвращает результат связанного с ним выражения. Синтаксис этих двух разновидностей выражений CASE:

```
Простое выражение Case :=
CASE выражение
WHEN результат_1 THEN
результирующее_выражение_1
WHEN результат_2 THEN
результирующее_выражение_2
...
ELSE
результирующее_выражение_else
END;
Поисковое выражение Case :=
CASE
WHEN выражение_1 THEN
результирующее_выражение_1
WHEN выражение_2 THEN
результирующее_выражение_2
...
ELSE
результирующее_выражение_else
END;
```

Выражение CASE возвращает одно значение — результат выбранного для вычисле- ния выражения. Каждой ветви WHEN должно быть поставлено в соответствие одно результирующее выражение (но не команда). В конце выражения CASE не ставится ни точка с запятой, ни END CASE. Выражение CASE завершается ключевым словом END. Выражение CASE может быть использовано в запросе:

```
SELECT s.v_name,
       s.b_add_service,
       CASE s.b_add_service
         WHEN 0 THEN
          'Основная'
         ELSE
          'Дополнительная'
       END
  FROM fw_service s;

SELECT s.v_name,
       s.b_add_service,
       CASE
         WHEN s.b_add_service = 0 THEN
          'Основная'
         ELSE
          'Дополнительная'
       END
FROM fw_service s;
```

Или как часть выражения, например при вызове процедуры:

```
P_BASE_EXCHANGE.saveServices(pID_SERVICE_INST    => dID_SERVICE_INST,
                             pDT_START           => pDATE_CUR,
                             pID_SERVICE         => i.id_srv,
                             pID_PRODUCT_INST    => pID_PRODUCT_INST,
                             pV_STATUS           => case
                                                      when i.v_ext_ident
= 'NET'
                                                       then
                                                       'A'
                                                      else
                                                       'W'
                                                    end,
                             pID_DEPENDENCY_INST => null);
```

14 Точка с запятой как разделитель

Программа на PL/SQL представляет собой последовательность объявлений и команд, которые определяются логически, а не физически — иначе говоря, их границы определяются не физическим завершением строки кода, а специальным завершите- лем — символом точки с запятой (;). Более того, одна команда нередко распространяется на несколько строк для удобства чтения. Например, следующая команда IF занимает три строки, а отступы более наглядно выделяют логику ее работы:

```
IF salary < min_salary(2003) THEN
    salary := salary + salary * .25;
  END IF;
```

В ней присутствуют два символа точки с запятой. Первый отмечает конец единствен- ной команды присваивания в конструкции IF-END IF, а второй — конец команды IF. Эту команду можно было бы записать в одной физической строке, результат будет одинаковым:

```
IF salary < min_salary (2003) THEN salary := salary + salary*.25; END IF;
```

Каждая исполняемая команда должна завершаться точкой с запятой, даже если она вложена в другую команду. Это относится также к командам create or replace procedure...is...begin...end; , case ... when ... then ... end case; , begin exception when ... then end; и другим составным командам.

15 Циклы

Чтобы дать начальное представление о разных циклах и о том, как они работают, рассмотрим три процедуры.
Простой цикл начинается с ключевого слова LOOP и завершается командой END LOOP. Выполнение цикла прерывается при выполнении команды EXIT, EXIT WHEN или RETURN в теле цикла (или при возникновении исключения).
Цикл WHILE имеет много общего с простым циклом. Принципиальное отличие заключается в том, что условие завершения проверяется перед выполнением очередной итерации
Цикл FOR существует в двух формах: числовой и курсорной. В числовых циклах FOR программист задает начальное и конечное целочисленные значения, а PL/SQL перебирает все промежуточные значения, после чего завершает цикл.
Курсорная форма цикла FOR имеет аналогичную базовую структуру, но вместо гра- ниц числового диапазона в ней задается курсор или конструкция SELECT.
Рассмотрим формирование пароля пользователя с помощью трех разных циклов:
– В первом случае мы просто перебираем целые числа от одного до восьми с помощью цикла for

```
FUNCTION get_pass(pLogin IN VARCHAR2) RETURN VARCHAR2 IS

l_password VARCHAR2(15);

BEGIN
  FOR i in 1 .. 8 LOOP
    l_password := l_password ||
    substr('23456789abcdefghjkmnpqrstuvwxyzABCDEFGHJKNPQRSTUVWXYZ',
                                        ceil(DBMS_RANDOM.value(0,
    length('23456789abcdefghjkmnpqrstuvwxyzABCDEFGHJKNPQRSTUVWXYZ'))),
                                        1);

  end loop;
  RETURN l_password;
END;
```

– Во втором случае мы выполняем цикл до тех пор, пока длина пароля менее 8 символов

```
FUNCTION get_pass(pLogin IN VARCHAR2) RETURN VARCHAR2 IS

l_password VARCHAR2(15);

BEGIN
  WHILE nvl(LENGTH(l_password),0) < 8 LOOP
    l_password :=  l_password ||
    substr('23456789abcdefghjkmnpqrstuvwxyzABCDEFGHJKNPQRSTUVWXYZ',
                                        ceil(DBMS_RANDOM.value(0,
    length('23456789abcdefghjkmnpqrstuvwxyzABCDEFGHJKNPQRSTUVWXYZ'))),
                                        1);

  end loop;
  RETURN l_password;
END;
```

– В третьем случае мы выполняем выход из цикла в случае, если длина пароля равна 8 символам.

```
FUNCTION get_pass(pLogin IN VARCHAR2) RETURN VARCHAR2 IS

l_password VARCHAR2(15);

BEGIN
  LOOP
    EXIT WHEN LENGTH(l_password) = 8
    l_password :=  l_password ||
    substr('23456789abcdefghjkmnpqrstuvwxyzABCDEFGHJKNPQRSTUVWXYZ',
                                       ceil(DBMS_RANDOM.value(0,
    length('23456789abcdefghjkmnpqrstuvwxyzABCDEFGHJKNPQRSTUVWXYZ'))),
                                       1);

  end loop;
  RETURN l_password;
END;
```

16 Команда CONTINUE

Используется для выхода из текущей итерации цикла и немедленного перехода к следующей итерации. Как и EXIT, эта команда существует в двух формах: безуслов- ной (CONTINUE) и условной (CONTINUE WHEN). Простой пример использования CONTINUE WHEN для пропуска итераций с четными значениями счетчика:

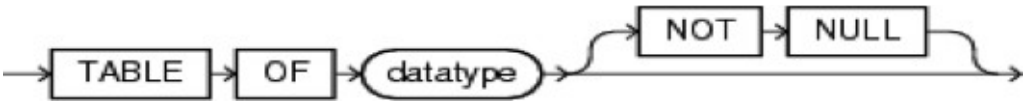
```
BEGIN
  FOR l_index IN 1 .. 10 LOOP
    CONTINUE WHEN MOD(l_index, 2) = 0;
    DBMS_OUTPUT.PUT_LINE('Счетчик = ' || TO_CHAR(l_index));
  END LOOP;
END;
```

17 Коллекции

Почти в любом языке программирования есть те или иные средства для работы с массивами (списками). В PL/SQL эти инструменты именуются «методы работы с коллекциями». Коллекции есть нескольких видов: ассоциативный массив, varray и nested table. Сначала поговорим о nested table, так как именно он чаще всего будет вам встречаться в начале. У коллекций есть несколько встроенных методов работы с ней:

-
    -
        -
            - DELETE - Удаляет элементы из коллекции;
            - TRIM - Удаляет элементы с конца коллекции (работает с внутренним размером коллекции);
            - EXTEND - Добавляет элементы в конец коллекции;
            - EXISTS - Возвращает TRUE, если элемент присутствует в коллекции;
            - FIRST - Возвращает первый индекс коллекции;
            - LAST - Возвращает последний индекс коллекции;
            - COUNT - Возвращает количество элементов в коллекции;
            - LIMIT - Возвращает максимальное количество элементов, которые может хра- нить коллекция;
            - PRIOR - Возвращает индекс предыдущего элемента коллекции;
            - NEXT - Возвращает индекс следующего элемента коллекции; NT поддерживает все методы, кроме LIMIT.

Объявление типа коллекции:



Datatype — это любой тип данных, кроме ref cursor. Приведём пример использования встроенных методов:

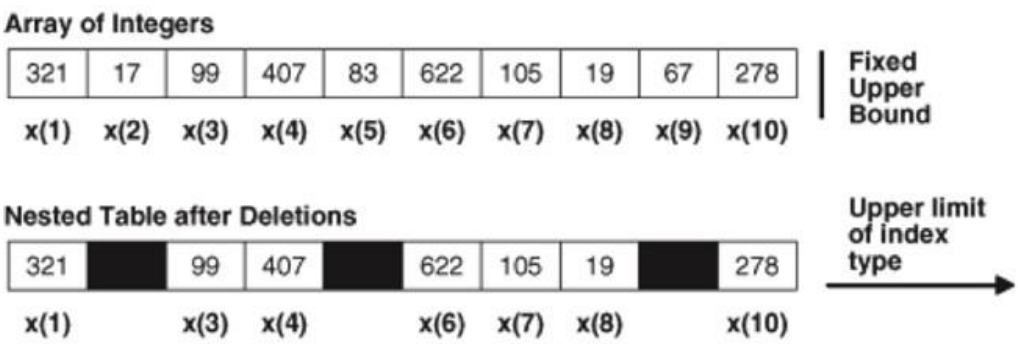
```
DECLARE
    --объявление типа коллекции
    TYPE nested_typ IS TABLE OF NUMBER;
    --объявление экземпляра коллекции
    nt1 nested_typ := nested_typ();

    ind number;

    procedure print_list(t in nested_typ) is
    begin
      for i in t.first .. t.last loop
        dbms_output.put_line(t(i));
      end loop;
    end;
BEGIN
      --
    dbms_output.put_line(nt1.count);
    nt1.extend;
    dbms_output.put_line(nt1.count);
    dbms_output.put_line(nvl(to_char(nt1(1)), 'NULL'));
    --
    nt1(nt1.last) := 6;
    nt1.extend; nt1(nt1.last) := 3;
    nt1.extend; nt1(nt1.last) := -1;
    nt1.extend; nt1(nt1.last) := -1;
    nt1.extend; nt1(nt1.last) := 7;
    --
    dbms_output.put_line('Состав коллекции 1:');
    print_list(nt1);
    --
    nt1 := nested_typ(3, 3, 7, -1, 2, 0, 5);
    dbms_output.put_line('Состав коллекции 2:');
    print_list(nt1);
    --
    ind := nt1.last;
    nt1.trim(2);
    dbms_output.put_line('Состав коллекции 3:');
    print_list(nt1);


    if nt1.exists(ind) then
      dbms_output.put_line('Элемент ' || to_char(ind) || ' есть в
коллекции');
    else
      dbms_output.put_line('Элемента ' || to_char(ind) || ' нет в
коллекции');
    end if;
END;
```

Для коллекций в PL/SQL возможна разреженность:



Наличие разреженности открывает новое поле для возможных обращений к эле- менту коллекции, которого не существует (ячейка памяти выделена, т.е. элемент объявлен, но не инициализирован). Поэтому код:

```
for ttab.first .. ttab.last loop
    ...
end loop;
\end{minted}

вообще говоря, корректнее, чем:
for 1 .. ttab.count loop
    ...
end loop;

и тем более корректней, чем (например):
for 1 .. 10 loop
    ...
end loop;
```

Всюду выше ttab - некоторая коллекция.

Кроме того, есть ещё удобный набор логических операций над коллекциями:

```
DECLARE
    --объявление типа коллекции
    TYPE nested_typ IS TABLE OF NUMBER;
    --объявление экземпляров коллекции
    nt1 nested_typ := nested_typ(1, 2, 3);
    nt2 nested_typ := nested_typ(3, 2, 1);
    nt3 nested_typ := nested_typ(2, 3, 1, 3);
    nt4 nested_typ := nested_typ();
BEGIN
    IF nt1 = nt2 THEN
        DBMS_OUTPUT.PUT_LINE('nt1 = nt2');
    END IF;

    IF (nt1 IN (nt2, nt3, nt4)) THEN
        DBMS_OUTPUT.PUT_LINE('nt1 IN (nt2,nt3,nt4)');
    END IF;

    IF (nt1 SUBMULTISET OF nt3) THEN
        DBMS_OUTPUT.PUT_LINE('nt1 SUBMULTISET OF nt3');
    END IF;

    IF (3 MEMBER OF nt3) THEN
        DBMS_OUTPUT.PUT_LINE('3 MEMBER OF nt3');
    END IF;

    IF (nt3 IS NOT A SET) THEN
        DBMS_OUTPUT.PUT_LINE('nt3 IS NOT A SET');
    END IF;

    IF (nt4 IS EMPTY) THEN
        DBMS_OUTPUT.PUT_LINE('nt4 IS EMPTY');
    END IF;
END;
```

К коллекции можно обращаться как к таблице внутри sql запросов. Допустим, на уровне схемы объявлен тип

```
create or replace type t_exchange_integer_table is table of number(10);
```

Тогда запрос к NT можно сделать с помощью следующей конструкции:

```
select column_value from table(t_exchange_integer_table(1, 2, 3));
```

Если нужно две коллекции соединить по их номеру элемента, то

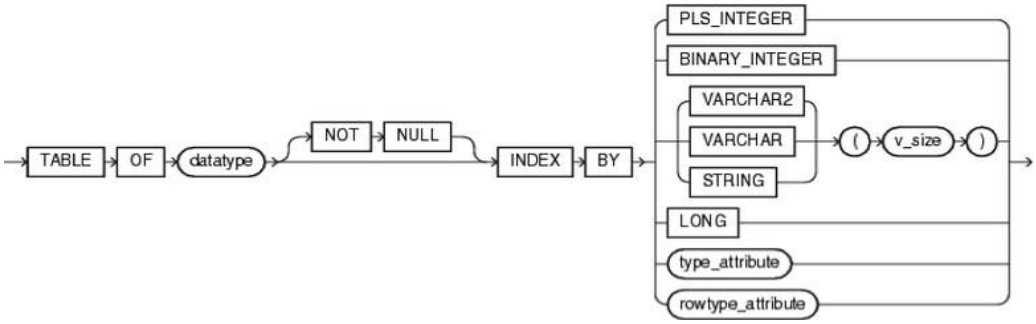
```
select t1.value, t2.value
  from (select column_value as value, rownum as id
          from table(t_exchange_integer_table(1, 2, 3))) t1
  join (select column_value as value, rownum as id
          from table(t_exchange_integer_table(3, 4, 5))) t2
    on t1.id = t2.id
```

18 Ассоциативный массив

Аналог словаря в python. Характеризуется следующим:

-
    - Набор пар ключ-значение
    - Данные хранятся в отсортированном по ключу порядке
    - Не поддерживает DML-операции (не может участвовать в селектах, не может храниться в таблицах)
    - При объявлении как константа должен быть сразу инициализирован функцией
    - Нельзя объявить тип на уровне схемы, но можно в пакете
    - Не имеет конструктора
    - Индекс не может принимать значение null (но допускает пустую строку)
    - Datatype – это любой тип данных, кроме ref cursor

Синтаксис объявления типа ассоциативного массива следующий:



В следующем примере мы объявляем тип ассоциативного массива, а затем коллекцию, основанную на этом типе. Коллекция заполняется четырьмя элементами, после чего мы перебираем ее содержимое и выводим символьные строки. Более подробное объяснение приведено в комментариях.

```
DECLARE
  subtype default_str is varchar2(255);
  /*Объявление ассоциативного массива TYPE с характерной секцией INDEX BY.
  Коллекция, созданная на основе этого типа, содержит список строк,
  каждая из которых может достигать по длине столбца 255 символов*/
  TYPE list_of_names_t IS TABLE OF default_str INDEX BY default_str;
  /*Объявление коллекции happystudents на базе типа list_of_names_t*/
  happystudents list_of_names_t;
  indx          default_str;
BEGIN
  /*Заполнение коллекции четырьмя именами. Обратите внимание:
  мы можем использовать любые строки меньше 255 символов в
  качестве индексов*/
  happystudents('Вова') := 'Бычков';
  happystudents('Саша') := 'Рыжиков';
  happystudents('Тест') := 'Тестов';
  happystudents('Олег') := 'Олег';
  indx := 'Олег';
  /*Проверка существования в нашей коллекции индекса indx равного 'Олег'*/
  if happystudents.exists('Олег') then
    dbms_output.put_line('Элемент с индексом ' || indx ||
                         ' есть в коллекции');
  end if;
  /*Вызов метода FIRST (функция, «прикрепленная» к коллекции)
  для получения первого (минимального) номера строки в коллекции*/
  indx := happystudents.FIRST;
  /*Перебор содержимого коллекции в цикле WHILE, с выводом каждой строки.
    Вызывается метод NEXT, который переходит от текущего элемента к
    следующему без учета промежуточных пропусков*/
  WHILE (indx IS NOT NULL) LOOP
    DBMS_OUTPUT.put_line('Индекс ' || indx || ' значение ' ||
                         happystudents(indx));
    indx := happystudents.NEXT(indx);
  END LOOP;
END;
```

Можно использовать и другие типы данных.

```
DECLARE
  /*Объявление ассоциативного массива TYPE с характерной секцией INDEX BY.
   Коллекция, созданная на основе этого типа, содержит список строк,
   каждая из которых может достигать по длине столбца 255 символов*/
  TYPE list_of_names_t IS TABLE OF varchar2(255) INDEX BY pls_integer;
  /*Объявление коллекции happystudents на базе типа list_of_names_t*/
  happystudents list_of_names_t;
  l_row         PLS_INTEGER;
BEGIN
  /*Заполнение коллекции четырьмя именами. Обратите внимание: мы можем
  использовать любые целочисленные значения по своему усмотрению.
  Номера строк в ассоциативном массиве не обязаны быть последовательными;
  они даже могут быть отрицательными! Никогда не пишите код с произвольно
  выбранными, непонятными значениями индексов! Этот пример всего лишь
  демонстрирует гибкость ассоциативных массивов*/
  happystudents(2020202020) := 'Александра';
  happystudents(-15070) := 'Анастасия';
  happystudents(-90900) := 'Дарья';
  happystudents(88) := 'Олег';
  /*Проверка существования в нашей коллекции индекса 88*/
  if happystudents.exists(88) then
    dbms_output.put_line(happystudents(88) || ' на месте');
  end if;
  /*Вызов метода FIRST (функция, «прикрепленная» к коллекции) для
получения
   первого (минимального) номера строки в коллекции*/
  l_row := happystudents.FIRST;
  /*Перебор содержимого коллекции в цикле WHILE, с выводом каждой строки.
    Вызывается метод NEXT, который переходит от текущего элемента к
    следующему без учета промежуточных пропусков*/
  WHILE (l_row IS NOT NULL) LOOP
    DBMS_OUTPUT.put_line('Индекс ' || l_row || ' значение ' ||
                         happystudents(l_row));
    l_row := happystudents.NEXT(l_row);
  END LOOP;
END;
```

19 Решение задачи

Сформулируем и решим задачу, с использованием полученных знаний.

> Создать ассоциативный массив, который будет содержать все дни недели и список их дат в 2019 году.

Сначала попробуем получить список всех дней недели:

```
/*Получим список дней недели*/


declare
  --подтип типа varchar2, в котором будет 255 символов
  subtype default_str is varchar2(255);
  --тип nested table для хранения дней недели
  type t_days is table of default_str;
  --переменная этого типа
  l_t_days t_days := t_days();
  --стартовая дата. может быть любой
  l_start_date date := to_date('2019-01-01', 'yyyy-mm-dd');
  --дата, которую мы будем использовать при продвижении по дням
  l_current_date date := l_start_date;
  --строка для хранения определенного дня
  day_of_week default_str;
begin
  --наполним табличку дней всеми днями недели и выведем на экран
  loop
    --получаем день недели
    day_of_week := to_char(l_current_date, 'day');
    --как только он уже есть в коллекции - выходим
    exit when(day_of_week member of l_t_days);
    --расширяем коллекцию
    l_t_days.extend;
    --присваиваем текущий день недели
    l_t_days(l_t_days.last) := day_of_week;
    --прибавляем ещё день
    l_current_date := l_current_date + interval '1' day;
  end loop;
  --выведем все в output
  DBMS_OUTPUT.put_line('NT no sort' || chr(10));
  for i in l_t_days.first .. l_t_days.last loop
    dbms_output.put_line(l_t_days(i));
  end loop;
  --если бы мы наполняли ассоциативный массив, а в качестве индекса брали
бы
  --to_char(date, 'd'), то у нас бы получилась упорядоченная коллекция
  declare
    --объявим ассоциативный массив дней недели с индексом в виде целых
чисел,
    --означающих порядковый номер дня и нужных для сортировки
    type t_dates_ind is table of default_str index by pls_integer;
    --переменную этого типа
    l_t_days_ind t_dates_ind;
    --переменную под индекс
    ind pls_integer;
    --этот тип и переменные видны только в данном блоке
  begin
    --сдвинем дату на начало(можно не делать, для чистоты результата)
    l_current_date := l_start_date;
    loop
      --получим порядковый номер дня
      ind := to_char(l_current_date, 'd');
      --если уже есть индекс равный порядковому номеру, то выходим
      exit when(l_t_days_ind.exists(ind));
      --получим название дня
      day_of_week := to_char(l_current_date, 'day');
      --добавим в ассоциативный массив получившуюся связку
      l_t_days_ind(ind) := day_of_week;
      --сдвинем счетчик
      l_current_date := l_current_date + interval '1' day;
    end loop;
    --выведем на экран
    ind := l_t_days_ind.FIRST;
```

```
    DBMS_OUTPUT.put_line(chr(10) || 'Ассоциативный массив' || chr(10));
    WHILE (ind IS NOT NULL) LOOP
      DBMS_OUTPUT.put_line(l_t_days_ind(ind));
      ind := l_t_days_ind.NEXT(ind);
    END LOOP;
  end;
  --Попробуем упорядочить обычную коллекцию. Рассмотрим два способа
  --В первом способе мы будем перебирать дни недели, пока не найдем
понедельник,
  --то есть первый порядковый номер
  --Этот вариант не оптимален,а нужен лишь для показа работы or в if
  declare
    ind number;
  begin
    l_t_days       := t_days();
    l_current_date := l_start_date;
    loop
      day_of_week := to_char(l_current_date, 'day');
      ind         := to_char(l_current_date, 'd');
      exit when(day_of_week member of l_t_days);
      if l_t_days.count = 0 or l_t_days(l_t_days.last) is not null then
        --заметим, что мы бы упали в случае, если при l_t_days.count = 0
        -- проверялись оба условия if. Но поскольку проверка "ленивая", то
все ок
        l_t_days.extend;
      end if;
      if l_t_days.last = ind then
        l_t_days(l_t_days.last) := day_of_week;
      end if;
      l_current_date := l_current_date + interval '1' day;
    end loop;
    DBMS_OUTPUT.put_line(chr(10) || 'NT' || chr(10));
    for i in l_t_days.first .. l_t_days.last loop
      dbms_output.put_line(l_t_days(i));
    end loop;
  end;
  --Во втором способе мы сразу расширим нашу коллекцию на семь элементов
  --под все дни недели.
  declare
    ind number;
  begin
    l_t_days       := t_days();
    l_current_date := l_start_date;
    --инициализируем все семь ячеек, но их значения пусты
    for i in 1 .. 7 loop
      l_t_days.extend;
    end loop;
    loop
      --получим название дня
      day_of_week := to_char(l_current_date, 'day');
      --получим порядковый номер
      ind         := to_char(l_current_date, 'd');
      --выходим когда повтор названия
      exit when(day_of_week member of l_t_days);
      --добавляем как в ассоциативный
      l_t_days(ind) := day_of_week;
      l_current_date := l_current_date + interval '1' day;
    end loop;
    --выводим
    DBMS_OUTPUT.put_line(chr(10) || 'NT2' || chr(10));
    for i in l_t_days.first .. l_t_days.last loop
      dbms_output.put_line(l_t_days(i));
    end loop;
  end;
```

```
  -- По времени выполнения первый, второй и четвертый будут примерно
равны,
  --а третий способ будет чуть дольше, засчет перебора лишних дней до
понедельника
  --кроме того, можно написать ещё один способ, который сразу сдвинет дату
  --на понедельник через NEXT_DAY, но он не добавит ничего нового
end;
```

Решим основную задачу, используя подзадачу:

```
/*Создать ассоциативный массив, который будет содержать
все дни недели и список их дат в 2019 году*/
declare
  --подтип типа varchar2, в котором будет 255 символов
  subtype default_str is varchar2(255);
  --тип nested table для хранения дней недели
  type t_days is table of default_str;
  --переменная этого типа
  l_t_days t_days := t_days();
  --тип nested table для хранения коллекции дат
  type t_dates is table of date;
  --переменная этого типа
  l_t_dates t_dates;
  --тип ассоциативный массив для хранения коллекций дат с индексами,
  --представляющими дни недели
  type t_day2dates is table of t_dates index by default_str;
  --переменная этого типа
  l_t_day2dates t_day2dates;
  --начало нашего отрезка времени
  l_start_date date := to_date('2019-01-01', 'yyyy-mm-dd');
  --дата, которую мы будем использовать при продвижении по дням
  l_current_date date := l_start_date;
  --строка для хранения определенного дня
  day_of_week default_str;
  --функция из предыдущей подзадачи
  function get_days return t_days is
    ind default_str;
  begin
    l_t_days      := t_days();
    l_current_date := l_start_date;
    for i in 1 .. 7 loop
      l_t_days.extend;
    end loop;
    loop
      day_of_week := to_char(l_current_date, 'day');
      ind         := to_char(l_current_date, 'd');
      exit when(day_of_week member of l_t_days);
      l_t_days(ind) := day_of_week;
      l_current_date := l_current_date + interval '1' day;
    end loop;
    return l_t_days;
  end;
  --функция для получения всех дат в формате dd.mm через запятую
  --из коллекции дат
  function date_collection2varchar2(date_collection in t_dates)
    return varchar2 is
    res varchar2(4000);
  begin
    --при движении по коллекции записываем в res предыдущее
    --значение и добавляем дату -с запятой
    for i in date_collection.first .. date_collection.last loop
      res := res || to_char(date_collection(i), 'dd.mm') || ', ';
    end loop;
    --при возврате обрезаем лишнюю запятую на конце
    return substr(res, 0, length(res) - length(', '));
  end;
begin
  --получаем коллекцию дней
  l_t_days := get_days;
  --инициализируем коллекции для каждого дня, чтобы их можно было
расширять
  --через extend
  for i in l_t_days.first .. l_t_days.last loop
    l_t_day2dates(l_t_days(i)) := t_dates();
  end loop;
```

```
  --переставляем l_current_date на начало года, ведь оно сдвинулось в
  --get_days попробуйте закомментировать эту строчку, и вы получите
  --дни начиная с 8 января. Это важно!
  l_current_date := l_start_date;
  --идем циклом по всем дням года
  while l_current_date < to_date('2020-01-01', 'yyyy-mm-dd') loop
    --получаем день недели
    day_of_week := to_char(l_current_date, 'day');
    --получаем из ассоциативного массива нашу коллекцию дат
    l_t_dates   := l_t_day2dates(day_of_week);
    --расширяем
    l_t_dates.extend;
    --присваиваем текущую дату
    l_t_dates(l_t_dates.last) := l_current_date;
    --заменяем коллекцию дат на новую
    l_t_day2dates(day_of_week) := l_t_dates;
    --можно и так, тогда не нужно заводить l_t_dates, но код менее
очевиден
    /* l_t_day2dates(day_of_week).extend;
    l_t_day2dates(day_of_week)(l_t_day2dates(day_of_week).last)
    := l_current_date;*/
    --прибавляем один день к счетчику
    l_current_date := l_current_date + interval '1' day;
  end loop;
  --выведем результат
  day_of_week := l_t_day2dates.FIRST;
  WHILE (day_of_week IS NOT NULL) LOOP
    DBMS_OUTPUT.put_line(day_of_week);

DBMS_OUTPUT.put_line(date_collection2varchar2(l_t_day2dates(day_of_week)))
;
    day_of_week := l_t_day2dates.NEXT(day_of_week);
  END LOOP;
end;
/*обратите внимание, что если бы нам нужно было вывести все
 даты одного дня недели, то достаточно было бы проходить
  через NEXT_DAY, а не по всем дням года*/
```