PL/SQL alapok

Doksi: PL/SQL Language Reference

Lexikális elemek: delimiterek, szimbólikus nevek (azonosítók, foglalt szavak), literálok, commentek

Delimiterek: egyszerű és összetett szimbólumok pl. "+" illetve ">=" ... stb.

Azonosítók: betűvel kezdődik (kötőjel, slash, szóköz tilos, $, #, aláhúzás lehet), de megengedett az úgynevezett idézőjeles azonosító pl. "A + B", ilyenkor a kisbetű/nagybetű nem ugyanaz.

Literálok: numerikus, karakter, karakterlánc, dátum, timestamp, logikai (`TRUE` és `FALSE` lehet) **(!)** a nulla hosszú karakterlánc literál NULL-nak felel meg.

Megjegyzések: Lehet egysoros (-- után) és több soros /* ... */ között

Címke: bármely végrehajtható utasítás címkézhető, pl. <<cimke>> v:=2;
Egyes vezérlőutasításoknál (GOTO, EXIT) és névhivatkozásoknál van jelentősége (címke.v := 2;)

Pragmák: a fordítónak szóló direktívák, feldolgozásuk fordítási időben történik.
`Pl. AUTONOMOUS_TRANSACTION, SERIALLY_REUSABLE, EXCEPTION_INIT`

Adattípusok
(skalár, összetett, LOB, referenciatípusok)
Lehet előre definiált, vagy a felhasználó által definiált.
Az adattípus meghatározza az értéktartományt, az elvégezhető műveleteket és a belső tárolást (reprezentációt). Az előre definiált típusokat a STANDARD csomag tartalmazza. Az adattípusokhoz megadhatunk **altípusok**at. Ezeknek a műveletei és reprezentációja azonos a főtípuséval, a tartományuk részhalmaza a főtípusénak. Ha nem valódi részhalmaz, akkor nem korlátozott altípusról (szinonimáról) van szó.

Numerikus típuscsalád:
NUMBER (hatékony tárolás, lassú műveletek -> konverzió előtte)
Altípusai: NUMERIC, DECIMAL, REAL, FLOAT, INTEGER stb.

BINARY_INTEGER (gyors műveletvégzés, könyvtári aritmetika)
Altípusai: PLS_INTEGER (még gyorsabb műveletek -> gépi aritmetika), NATURAL

BINARY_FLOAT (gépi aritmetika)
BINARY_DOUBLE (gépi aritmetika)

Karakteres típuscsalád:
CHAR[(hossz) | [byte | char]] hossz -> 1..32767, alapértelmezés: char
VARCHAR2 (mint a char, altípusok: string, varchar)
NCHAR, NVARCHAR2 (a hossz mindig karakterekben értendő)
ROWID, UROWID

Dátum típuscsalád:
DATE
TIMESTAMP
INTERVAL

Logikai típuscsalád:
BOOLEAN

Összetett típusok: record, table, array
LOB típusok: bfile, clob, blob, nclob
Ref típusok: ref cursor, sys_refcursor, ref obj_típus

Felhasználói altípusok megadása:
SUBTYPE altípus_neve IS alaptípus;

```
Pl.  SUBTYPE t_evszam IS NUMBER(4,0);
     SUBTYPE t_nev    IS VARCHAR2(30);
```

Adatkonverzió:
Amennyiben lehetséges, implicit konverziót végez. A legfontosabb konverziós függvények:
`TO_CHAR, TO_NUMBER, TO_DATE, TO_TIMESTAMP, TO_DSINTERVAL, TO_YMINTERVAL, CHARTOROWID`

Kifejezések:
Operandusokból és operátorokból áll. Operandus lehet literál, konstans, változó, fv-hívás.
Műveletek precedenciája (csökkenő sorrendben):

(**, NOT) hatványozás, logikai tagadás
(+, -) azonosság, negatív
(*, /) szorzás, osztás
(+, -, ||) összeadás, kivonás, konkatenáció
(=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN)
(AND) és
(OR) vagy

(!) használjunk zárójeleket

Feltételes kifejezések

```
CASE szelektor
  WHEN kifejezés THEN eredmény
  WHEN kifejezés THEN eredmény
  ELSE eredmény
END
```

vagy

```
CASE  WHEN logikai_kifejezés THEN eredmény
      WHEN logikai_kifejezés THEN eredmény
      ELSE eredmény
END
```

A fenti konstrukciók mindenhol előfordulhatnak, ahol kifejezések megengedettek, pl. egy SELECT utasításban is. Példák:

```
select fizetes, case when fizetes > 2500 then 'sok'
                     when fizetes > 2000 then 'ok'
                     else 'kicsi'
                end
from dolgozo;

select foglalkozas, case foglalkozas
                       when 'MANAGER'  then 'főnök'
                       when 'SALESMAN' then 'ügynök'
                       else 'más'
                    end
from dolgozo;
```

(!) A fentiek nem keverendők össze a CASE utasítással.

Deklarációk:
PL/SQL blokk, alprogram vagy package deklaratív részében adhatók meg.
Fontos (!) beágyazott blokknak is lehet deklaratív része. Példa az alábbi:

```
<<cimke1>>
DECLARE
  a  NUMBER;
BEGIN
  a:=2;
  <<cimke2>>
  DECLARE
    a number;
  BEGIN
    a:=4;
    dbms_output.put_line(cimke1.a);
    dbms_output.put_line(cimke2.a);
    dbms_output.put_line(a);
  END;
  dbms_output.put_line(a);
END;
```

Futtatás előtt SET SERVEROUTPUT ON [SIZE n]
Ennek hatására íródik ki képernyőre a dbms_output.put_line() kimenete.

Értékadás a deklarációval egyidejűleg:

```
  v   BOOLEAN  :=  FALSE;
```

vagy ezzel egyenértékű az alábbi:

```
  v   BOOLEAN DEFAULT FALSE;
```

NOT NULL is megadható a deklarációval egyidejűleg, de ilyenkor kötelező a kezdeti értékadás.

```
  v   NUMBER  NOT NULL := 2;
```

Konstans deklaráció (ilyenkor is kötelező az értékadás)

```
v  CONSTANT  NUMBER  := 2;
```

Érvényesség és láthatóság:
Egy azonosító érvényes abban a blokkban, amelyben deklarálták. Ezen belül ott látható, ahol nem takarja el másik azonosító. Erre jó példa a fenti két egymásba ágyazott blokk.

Nevek feloldása:
Lokális változók elsőbbséget élveznek a táblanevekkel szemben, de az oszlopnevek elsőbbséget élveznek a lokális változókkal szemben.

Pl. UPDATE emp SET ... -- rossz ha van emp nevű változó is.
Megoldás a minősítés. Minősíteni lehet cimkével vagy alprogram nevével.

%TYPE (típus öröklés)
Két legfőbb előnye: 1. Nem kell ismernem az objektum típusát
2. Ha később változik a típus, a kód maradhat

Példa:
```
v    tablanev.oszlopnev%TYPE;
```

%ROWTYPE (sortípus/rekordtípus öröklés)
A deklarációban nem szerepelhet inicializáció, de rekordok közötti értékadás megengedett.
Példa:
```
rek1 := rek2;
rec     tablanev%ROWTYPE;
rec2    kurzornev%ROWTYPE;
```

Rekordok: (Lehet a %ROWTYPE segítségével és lehet saját rekordtípussal)

Rekord típus definiálása:
```
  TYPE  rekord_típus_neve  IS  RECORD (mező1, mező2 ...)
```
ahol a mezők megadása a következő:
```
  mezőnév   adattípus   [[NOT NULL] { :=| DEFAULT} kifejezés ]
```
megadhatunk beágyazott rekordokat is

Rekord deklarálása:
```
  rekord_név   rekord_típus_neve;
```

Hivatkozás rekord mezőire:
rekord_név.mező
beágyazott rekord esetén: rekord_név.mező.mező2
fv által visszaadott rekord esetén: fv(paraméter).mező
(!!!) paraméter nélküli fv esetén nem használható a fenti jelölés pl. fv().mező -- rossz

Példa:
```
DECLARE
  TYPE rektip IS RECORD(m1 INTEGER, m2 VARCHAR2(10));
  rec rektip;
BEGIN
  rec.m1 := 1; rec.m2 := 'Bubu';
  DBMS_OUTPUT. PUT_LINE(rec.m2);
END;
```

Végrehajtható utasítások

Üres utasítás:
```
NULL;
```

Értékadás:
```
v := kifejezés;
SELECT (FETCH) INTO v1, v2, … FROM …;
```

Példa:

```
v := 'Blabla';
SELECT oszlop1, oszlop2 INTO v1, v2 FROM … ;
SELECT oszlop1, oszlop2 INTO rec FROM …;
FETCH kurzornev INTO v1, v2;
FETCH kurzornev INTO rec;
```

GOTO

címke utáni utasításra ugrik (címke csak végrehajtható utasítás előtt lehet, így nem lehet pl. az END LOOP előtt, de ilyenkor segíthet a NULL utasítás.)
IF-be, ciklusba és blokkba nem lehet belépni vele
kivételkeelőből nem lehet az aktuális blokkba lépni
A blokkból a külső blokkba lehet lépni

Elágazás

```
  IF ... THEN
    utasítás1
  ELSIF ... THEN
    utasítás2
  ELSIF ... THEN
    utasítás3
  END IF;
```

(!) END IF; külön írni, utána pontosvessző
(!) ELSIF egybeírni és hiányzik belőle egy "E" betű (nem ELSEIF)

CASE utasítás

```
CASE
  WHEN feltétel THEN utasítások
  WHEN feltétel THEN utasítások
  ELSE utasítások
END CASE;

CASE szelektor
  WHEN kifejezés THEN utasítások
  WHEN kifejezés THEN utasítások
  ELSE utasítások
END CASE;
```

(!) nem keverendők össze a CASE feltételes kifejezéssel

Ciklusok

1. forma: (végtelen ciklus kilépés EXIT-tel.)

```
LOOP
  utasítások
END LOOP;
```

EXIT-tel kilépni csak ciklusból lehet, PL/SQL blokkból nem (-> RETURN)
EXIT másik formája az `EXIT WHEN feltétel`

Ciklusoknak címke adható, hasonlóan mint a PL/SQL blokkoknak

```
<<címke1>>
LOOP
  utasítások
  LOOP
    EXIT cimke1
  END LOOP;
END LOOP;
```

A fenti módon címke segítségével beágyazott ciklusok mélyéről is kiléphetünk.

2. forma

```
WHILE feltétel LOOP
  utasítások
END LOOP;
```

3. forma

```
FOR számláló IN [REVERSE] alsó..felső LOOP
  utasítások
END LOOP;
```

Az alsó és felső határ lehet literál, változó vagy kifejezés, de kiértékelés után egésznek kell lennie. A lépésköz csak 1 lehet. Az utóbbi két ciklusnak is adható címke és EXIT-tel ki is lehet lépni belőlük.

EXIT

```
EXIT [címke] [WHEN feltétel];
```

RETURN

Alprogram vagy blokk futásának befejezése.

SQL utasítások PL/SQL-ben

```
SELECT … INTO v1,v2 FROM
DELETE … RETURNING …
INSERT … RETURNING
UPDATE … RETURNING
```

Tranzakció-kezelő utasítások `(COMMIT, ROLLBACK, SAVEPOINT)`

Autonóm tranzakciók

Nem lehet DDL utasítást kiadni PL/SQL blokkban

Programegységek

A blokk

```
DECLARE
  deklarációk
BEGIN
  utasítások
EXCEPTION
  kivételkezelő
END;
```

A blokkok egymásba ágyazhatók. A blokk szerepelhet bárhol a programban, ahol utasítás állhat. A deklarációs részben deklarálható típus, változó, konstans, kivétel, kurzor, alprogram. Az alprogram deklarációk egy blokk deklarációs részének legvégén lehetnek csak az egyéb deklarációk után. A RETURN utasítás hatására a blokk futása befejeződik, sőt az őt tartalmazó blokkok futása is.

Alprogramok (procdúrák, fv-ek)

```
PROCEDURE  p_név(param) IS ... BEGIN ... END;
FUNCTION  f_név(param) RETURN típus IS ... BEGIN ... END;
```

Mindkettő két részből áll specifikációból és body-ból. A specifikáció az IS kulcsszóig tart. A body-nak van deklarációs, végrehajtható és kivételkezelő része. Visszatérés az alprogramból: RETURN (fv esetén visszatérési érték is kell)

A paraméterek megadása a következőképpen néz ki:

```
  p_név [{IN|OUT|IN OUT} [NOCOPY]] típus [{:=|DEFAULT} kifejezés]
```

A típus nem tartalmazhat korlátozásokat (hossz, pontosság, skála). A paraméter így lehet NUMBER, de nem lehet NUMBER(3), és nem lehet NOT NULL megszorítás a paraméterre.

Paraméterátadás módok: (IN, OUT, IN OUT)
IN esetén az aktuális paraméter kifejezés lehet, a másik két módnál csak változó. Az alprogram törzsében az IN módú paraméter konstansként, az OUT paraméter inicializálatlan változóként, az IN OUT pedig inicializált változóként kezelhető.
IN mód esetén a formális paraméter az aktuális paraméter értékének címét kapja meg, a másik két mód esetén az értékét. Ezt az értékmásolást letilthatjuk a NOCOPY (fordítónak szóló) opcióval, amit azonban az nem biztos, hogy figyelembe vesz. Ha nincs értékmásolás, az gyorsabb futást eredményezhet.
Az alprogram sikeres befejeződése esetén a formális paraméter értéke (amit az alprogram adott neki) visszamásolódik az aktuális paraméterbe. Sikertelen befejeződés esetén (pl. le nem kezelt kivétel esetén) ez a visszamásolás elmarad. NOCOPY opció esetén az alprogram referenciát kap meg, és minden általa megtett módosítás megőrződik akár sikeres, akár sikertelen volt a lefutás (lásd pl_nocopy példákat).

A függvény törzsében legalább egy RETURN utasításnak szerepelnie kell. A függvény mellékhatásának hívjuk, amikor a függvény megváltoztatja a paramétereit (OUT és IN OUT) vagy a környezetét (adatbázist, globális változókat).

Paraméterátadás pozíció illetve név szerint történik. fv(p_név => érték). A kettő keverhető is, de ilyenkor elöl szerepelnek a pozíció szerint átadottak. Az aktuális paraméterek száma lehet kevesebb a formális paraméterek számánál, amennyiben a többi formális paraméternek van kezdőértéke (DEFAULT). Lokális (nem tárolt) és package-beli alprogramok túlterhelhetők, ha a paraméterek száma vagy típusa eltérő.

```
DECLARE
  PROCEDURE elj(p IN NUMBER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('number param');
  END elj;

  PROCEDURE elj(p IN VARCHAR2) IS
  BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('varchar2 param');
  END elj;
BEGIN
  elj(100);
  elj('100');
END;
```

Mivel a plsql-ben minden nevet deklarálni kell a használat előtt, az alprogramoknál lehetőség van az előre deklarálásra is, hogy egymást kölcsönösen hívó alprogramokat tudjunk írni.

Beépített függvények
A beépített függvények a STANDARD csomagban vannak deklarálva, és ezek minden programban használhatók. Nem keverendők össze az SQL függvényekkel. A beépített függvények elsősorban procedurális utasításokban használhatók és nem SQL utasításokban. Azért a legtöbb használható SQL utasításban is, de van ami nem, pl. SQLCODE, SQLERRM. Ami viszont nem beépített függvény az nem használható csak SQL utasításban pl. az aggregátor fv-ek.

```
SELECT MAX(o) INTO valtozo FROM tábla;  -- Ez így rendben
valtozo := MAX(o);  -- Ez így hibás

SELECT SYSDATE INTO valtozo FROM dual;  -- Ez így rendben
valtozo := sysdate;  -- Ez is rendben
```

Kivételkezelés
Ha valami olyan dolog történik futás közben, ami megsérti az Oracle szabályait akkor a rendszer egy hibakódot és egy hibaeseményt generál. Ezeket kezelhetjük le a hibakezelő részben, ami a blokk végén szerepelhet.

Nélkülük minden utasítás után ellenőrizni kellene, hogy nem volt-e hiba. (pl. C-be ágyazásnál ezt tettük) Így elég egyszer megírni a hibaellenőrzést a blokk végén.
Viszont ha egy helyen van több lehetséges hiba kezelése akkor nem tudjuk pontosan, hogy hol merült fel a hiba.

A hibakezelő részben név szerint hivatkozhatunk a hibaeseményre, így csak olyan hibát tudunk lekezelni, aminek van neve. A felhasználó is létrehozhat névvel ellátott eseményt és vannak olyan hibaesemények, amiknek már van neve. (előre definiált hibaesemények)

Kivételek deklarálása:
```
  kivétel_név   EXCEPTION;
```
A kivételekre ugyanazok az érvényességi szabályok vonatkoznak mint a változókra. Egy blokkban deklarált kivétel a blokkra nézve lokális, címkézett blokk esetén hivatkozhatunk a kivételre `címke.kivétel_név` módon ... stb.

Az előre definiált kivételek a **STANDARD package**-ben vannak deklarálva az alábbi módon.

```
  CURSOR_ALREADY_OPEN exception;
     pragma EXCEPTION_INIT(CURSOR_ALREADY_OPEN, '-6511');
```

Nem minden belső hibának van neve. Ezeket nem tudjuk lekezelni, hacsak nem adunk nekik nevet. Ezt egy fordítónak szóló direktívával tehetjük meg, aminek a neve EXCEPTION_INIT.

```
  pragma EXCEPTION_INIT(hiba_név, hibakód);
```

A STANDARD package-ben ilyen direktívák is szerepelnek a kivételek deklarációja után.

A legfontosabb előre definiált kivételek a következők:

```
 CURSOR_ALREADY_OPEN exception;
 DUP_VAL_ON_INDEX exception;
 TIMEOUT_ON_RESOURCE exception;
-TRANSACTION_BACKED_OUT exception;
 INVALID_CURSOR exception;
 NOT_LOGGED_ON exception;
 LOGIN_DENIED exception;
 NO_DATA_FOUND exception;
 ZERO_DIVIDE exception;
 INVALID_NUMBER exception;
 TOO_MANY_ROWS exception;
 STORAGE_ERROR exception;
 PROGRAM_ERROR exception;
 VALUE_ERROR exception;
```

A fentieket a következő utasítással listázhatjuk ki a rendszer katalógusból:

```
  SELECT text FROM all_source WHERE type = 'PACKAGE'
  AND name = 'STANDARD' AND lower(text) LIKE '%exception_init%';
```

A kivételek meghívása:
A belső hibákat a rendszer automatikusan meghívja, ha előfordul az esemény, és ha névvel láttuk el őket, akkor ez egyben az adott nevű hibaesemény előfordulását is jelenti.
Az általunk deklarált kivételeket explicit módon meg kell hívni az alábbi módon:

```
  RAISE hibanév;
```

A fenti módon előre definiált (és névvel ellátott) eseményt is meghívhatunk.

Kivételek lekezelése:
A kivétel hívásakor (explicit vagy implicit módon) a vezérlés az aktuális blokk kivételkezelő részére adódik. Ha ott nincs lekezelve a kivétel akkor a külső blokknak adódik tovább, addig, amíg valahol le lesz kezelve. (ellenkező esetben hibaüzenet a futtató környezetnek)

```
   EXCEPTION
     WHEN kivétel_név OR kivétel_név2 THEN
       utasítások
```

A WHEN OTHERS megadásával minden hibát lekezelhetünk (a név nélkülieket is).

Néhány apró tudnivaló:
A deklarációban felmerülő hibákat rögtön a külső blokk fogja megkapni.
Nyitott kurzor esetén felmerülő hiba lekezelése előtt a kurzor automatikusan bezáródik, így arra hivatkozni nem lehet a hibakezelőben.
Ha a kivételkezelő részben felmerül egy hiba akkor a vezérlés rögtön a külső blokk hibakezelő részére adódik.

Hiba lekezelése majd továbbadása a külső blokknak -> RAISE; (kivételnév nélkül) Ilyen formában csak a kivételkezelőben fordulhat elő a RAISE utasítás.
Mi legyen ha a SELECT INTO nem ad vissza egyetlen sort sem? (`NO_DATA_FOUND`)
Megoldás -> alblokkba írás, amihez kivételkezelőt is írunk

Alprogramok hibáinak lekezelése:
A DBMS_STANDARD package raise_application_error(hibakód, hibaüzenet) procedúrájával az alprogramokból úgy térhetünk vissza, hogy egy megfelelő hibakódot adunk vissza a hívónak, amit az lekezelhet, ha a deklarációjában adott neki egy nevet. A megadható hibakódok –20000 és –20999 között kell hogy legyenek.
E nélkül csak a WHEN OTHERS résszel tudnánk lekezelni az alprogram hibáit, és így nem tudnánk megállapítani a hiba fajtáját.

SQLCODE és SQLERRM fv-ek
A felhasználó által definiált hibára +1-et ad vissza az SQLCODE, a belső hibákra pedig negatív számot. (kivétel +100 -> NO_DATA_FOUND) Az SQLERRM az éppen felmerült hiba üzenetét adja vissza, illetve a paramétereként megadott negatív számhoz, mint hibakódhoz tartozó hibaüzenetet. Így pl. az alábbi módon meg is tudjuk nézni a rendszer hibaüzeneteit:

```
FOR err_num IN 1..9999 LOOP
  dbms_output.put_line(SQLERRM(-err_num));
END LOOP;
```

A fenti két függvény nem használható közvetlenül SQL utasításban, pl.
`INSERT INTO tábla VALUES(SQLCODE);` nem működik,
helyette az értéküket lokális változóba kell tenni, majd úgy használni.

```
err_num := SQLCODE;
INSERT INTO tábla VALUES(err_num);
```

A le nem kezelt hibák esetén a rendszer különbözően viselkedik a futtató környezettől függően. Pl. egy C-be ágyazott program ROLLBACK-et ad ki, egy alprogram viszont nem, és az OUT típusú változóinak sem ad értéket, ahogyan korábban láttuk.

Kurzorok
Egy SQL utasítás soronkénti feldolgozásához használható. A kurzorok kezelése 4 lépésben valósul meg, ezek: 1. deklaráció, 2. megnyitás, 3. olvasás, 4. lezárás

```
1. CURSOR c_név [(paraméterek)] IS SELECT ...
2. OPEN c_név [(aktuális_paraméterek)]
3. FETCH c_név INTO változó_lista
4. CLOSE c_név
```

A fenti 4 lépés szintaktikailag egyszerűbb formában is megadható FOR ciklus segítségével.
Második megadási forma (implicit megnyitás, olvasás, lezárás)

```
  Deklaráció + FOR c_rec IN c_név LOOP ... END LOOP;
```

Harmadik megadási forma (nem kell deklaráció sem)

```
 FOR c_rec IN (SELECT ... ) LOOP ... END LOOP;
```

Az első két megadási formánál paraméterek adhatók meg a kurzornév után `c_név(param)` formában. A deklarációban adhatjuk meg a formális paramétereket, a megnyitásnál (illetve FOR ciklusnál) pedig az aktuális paramétereket.
A második és harmadik formánál a ciklusváltozó (c_rec) egy implicit módon deklarált rekord típusú változó. Így a hivatkozás rá c_rec.oszlop.

Kurzor attribútumok

```
%FOUND, %NOTFOUND, %ISOPEN, %ROWCOUNT
```

Ugyanezek az attribútumok implicit kurzor attribútumaként is használhatók, ekkor a legutóbbi SQL utasításra vonatkozóan adnak információt. Formájuk: `SQL%FOUND …`

Példa:

```
DECLARE
  CURSOR curs1 IS SELECT oazon, dnev FROM dolgozo WHERE oazon = 10;
  rec curs1%ROWTYPE;
BEGIN
  OPEN curs1;
  LOOP
    FETCH curs1 INTO rec;
    EXIT WHEN curs1%NOTFOUND;
    dbms_output.put_line(to_char(rec.oazon)||' - '||rec.dnev);
  END LOOP;
  CLOSE curs1;
END;
```

Módosítás/törlés kurzor sorain végighaladva

```
deklaráció -> (CURSOR c_név IS SELECT ... FOR UPDATE)
```

Majd a megnyitás és olvasás után az aktuális sor módosítása:

```
UPDATE tábla SET oszlop=... WHERE CURRENT OF c_név;
```

Illetve törlése:

```
DELETE FROM tábla WHERE CURRENT OF c_név;
```

Kurzor típusú változók

A változó egy kurzorra mutat, az OPEN utasításkor fogjuk megadni a lekérdezést. A változó lehet például alprogram paramétere is.

Kurzor típus definiálása

```
TYPE  cursor_tipus  IS  REF CURSOR [RETURN rekordtípus]
```

A rekordtípus megadása történhet %TYPE, %ROWTYPE, vagy saját rekordtípussal. Van egy generikus rekordtípus is, amikor nem adjuk meg a RETURN részt.

Változó deklarálása

```
c_változó  cursor_tipus
```

Kurzor megnyitása, olvasása, lezárása

```
 OPEN c_változó FOR SELECT ...
 FETCH c_változó INTO ...
 CLOSE c_változó
```

Ha a változót paraméterül adjuk át egy alprogramnak, amelyik megnyitja vagy lezárja a kurzort, akkor a paraméter IN OUT kell hogy legyen. Ha a paraméterül kapott kurzor változó nem megfelelő típusú akkor a ROWTYPE_MISMATCH hibát generálja a rendszer.

Package-ben nem deklarálhatunk REF CURSOR típusú változót, mert ezek a változók nem maradnak életben a session egész időtartama alatt, ellentétben a többi típusú változóval.

Tárolt alprogramok

A korábban látott alprogramok lokálisak voltak, és csak abban a blokkban voltak használhatók, amelyben deklaráltuk őket. A tárolt alprogramok ezzel szemben az adatbázisban tárolódnak.

```
CREATE OR REPLACE eljárásfej
[AUTHID {DEFINER|CURRENT_USER}]
eljárástörzs
```

vagy

```
CREATE OR REPLACE függvényfej
[AUTHID {DEFINER|CURRENT_USER}]
[DETERMINISTIC]
eljárástörzs
```

Az AUTHID segítségével megadható, hogy az eljárás létrehozójának vagy az aktuális hívójának jogosultságai legyenek-e érvényben a híváskor. Függvény esetén megadható a DETERMINISTIC kulcsszó, ami egy optimalizálási előírás, és a redundáns függvényhívások elkerülését szolgálja.

Alprogram újra fordítása:

```
ALTER FUNCTION  nev COMPILE [DEBUG];
ALTER PROCEDURE nev COMPILE [DEBUG];
```

Ahhoz hogy egy függvényt SQL utasításban is használhassunk, az alábbi megszorításoknak kell eleget tennie:

1. tárolt fv legyen
2. egy sorra vonatkozó legyen és ne egy csoportra
3. csak IN módú paraméterei legyenek
4. paramétereinek típusa Oracle belső típus legyen, és ne PLSQL típus
5. a visszaadott értékének típusa Oracle belső típus legyen

Package-ek

A package-ben lehetnek procedúrák, függvények, típus definíciók, változó deklarációk, konstansok, kivételek, kurzorok.

Két része a specifikációs rész és a törzs (body). A specifikációs részben vannak a publikus deklarációk. Ennek létrehozása (SQL utasítással):

```
CREATE OR REPLACE PACKAGE p_név IS
  publikus típus és objektum deklarációk
  alprogram specifikációk
END;
```

A body-ban vannak az alprogramok és a kurzorok implementációi. Csak ezeknek van implementációs része, így ha a package csak más objektumokat tartalmaz (változók, típusok, kivételek ... stb.) akkor nem is kell hogy body-ja is legyen.

A kurzorok kétféleképpen is megadhatók.

1. Vagy a specifikációban adjuk meg őket a szokásos módon, ekkor nem is szerepelnek az implementációs részben.
2. A specifikációs részben csak a nevét és a sortípusát adjuk meg (`CURSOR C1 RETURN sortípus`) és az implementációs részben adjuk meg a SELECT-et.

```
CREATE OR REPLACE PACKAGE BODY p_név IS
  privát típus és objektum deklarációk
  alprogramok törzse (PROCEDURE ... IS ...)
  kurzorok (CURSOR C1 RETURN sortípus IS SELECT ...)
[BEGIN  inicializáló utasítások ]
END;
```

A body-ban vannak az implementációk és lehet neki inicializációs része is (BEGIN ... END között), ami csak egyszer fut le, amikor a package-re először hivatkoznak.

A package specifikációs részében szereplő objektumok lokálisak az adatbázissémára nézve és globálisak a package-re nézve. **hivatkozás package-beli objektumokra:** p_név.obj
a STANDARD package-beli objektumokra hivatkozhatunk a p_név nélkül.

Lehet azonos a neve két package-ben levő alprogramnak, amelyeknek más a paraméterezése. Ilyenkor híváskor derül ki, hogy melyik fog futni a formális és aktuális paraméterek egyeztetésekor (túlterhelés). Például a STANDARD package-ben van több verzió is a TO_CHAR fv-re.

A package-ek legfontosabb előnyei:
Modularítás
Információ elrejtés
Egészben töltődik be a memóriába minden objektuma az első hivatkozáskor.
A package-ben deklarált változók és kurzorok a session végéig léteznek, így közösen használhatják azokat a többi programok. (Kivétel a REF CURSOR, ami package-ben nem deklarálható.)
Túlterhelt alprogramok írhatók (a lokális alprogramok is túlterhelhetők, csak a tároltak nem)

A package-ek forrásszövege a DBA_SOURCE táblában megnézhető.

A legfontosabb package-ek:

STANDARD	Beépített függvények és alprogramok ebben vannak
DBMS_SQL	DDL és dinamikus SQL végrehajtására
DBMS_OUTPUT	pl. put_line()
DBMS_STANDARD	az alkalmazás és az Oracle közötti interakciót segíti
UTL_FILE	op. rendszer fájlok írása, olvasása

Kollekciók
Ezek a szerkezetek a programozási nyelvek tömbjeihez hasonlóak. Háromféle kollekciót kezel a plsql, a beágyazott táblát, a dinamikus tömböt és az asszociatív tömböt. A kollekciók mindig egydimenziósak, de elemei lehetnek maguk is kollekciók. Az asszociatív tömb csak plsql programban használható, a másik két kollekció típus létrehozható adatbázis objektumként a CREATE TYPE utasítással, és adatbázistábla oszlopában is tárolható.
A kollekciók egész értékekkel indexelhetők, kivéve az asszociatív tömböt, amely VARCHAR2 típusú adattal is indexelhető. Az indexek 1-től indulnak. A dinamikus tömbben nem lehetnek lyukak, legfeljebb egyes elemek NULL értékűek.
A kollekciótípusokat először deklarálni kell:

Asszociatív tömb
```
TYPE t_név IS TABLE OF adattípus [NOT NULL] INDEX BY indextípus;
```
Az indextípus lehet BINARY_INTEGER, PLS_INTEGER, VARCHAR2, vagy ezek altípusa.

Beágyazott tábla
```
TYPE t_név IS TABLE OF adattípus [NOT NULL];
```

Dinamikus tömb
```
TYPE t_név IS VARRAY(méret) OF adattípus [NOT NULL];
```

A dinamikus tömb és a beágyazott tábla tulajdonképpen speciális objektumtípusok. A deklarációval egy referencia típusú változó jön létre, aminek automatikusan NULL a kezdőértéke. Inicializálni a típus konstruktorával tudjuk őket. Az asszociatív tömböket viszont nem lehet inicializálni.
A kollekciók lehetnek függvények paraméterei is, vagy a függvény által visszaadott érték is lehet kollekció típusú.

Kollekció metódusok

```
EXISTS(n) boolean -> létezik-e az n-edik elem
COUNT     number  -> hány nem törölt eleme van a kollekciónak
LIMIT     number  -> VARRAY maximális mérete, a másik kettőre NULL-t ad
FIRST     indextípus a legelső index értékét adja vissza, (NULL-t, ha üres)
LAST      indextípus az utolsó index értékét adja vissza, (NULL-t, ha üres)
PRIOR(n)  indextípus az n index előtti index (NULL ha nincs előtte index)
NEXT(n)   indextípus az n index utáni index (NULL ha nincs utána index)
EXTEND(n[,m])        bővíti a dinamikus tömböt vagy beágyazott táblát
  n elemmel bővíti, NULL értékekkel, illetve az m-edik elemet teszi be n-szer
TRIM(n)              eltávolítja din. tömb vagy beágyazott t. utolsó elemeit
DELETE(n)            az n-edik elemet törli
DELETE(m, n)         m-től n-ig törli az elemeket
DELETE               az összes elemet törli. Din tömbre csak így használható
```

Amíg egy kollekcióelemnek nem adtunk értéket addig az nem létezik. Ha hivatkozunk rá akkor a NO_DATA_FOUND kivételt generálja a rendszer.

BULK COLLECT és FORALL
A procedurális utasításokat a plsql motor hajtja végre, az SQL utasításokat azonban az sql motor. Így ha például ciklusban, egy kollekció elemeire hajtunk végre azonos utasítást, az nagyon sok motorváltást eredményez. Ezt kerülhetjük el az úgynevezett együttes hozzárendeléssel. Ennek két formája a FORALL és a BULK COLLECT. Az első esetén az egész kollekciót megkapja az SQL motor, a második esetén a lekérdezés teljes eredményét egyszerre kapja meg a plsql motor.
Példa:
```
FORALL i IN kollekció.COUNT
    UPDATE tábla SET oszlop = kifejezés WHERE oszlop = kollekció(i);
```
A FORALL esetén az SQL utasítás lehet INSERT, DELETE és UPDATE

A BULK COLLECT rész a SELECT, INSERT, DELETE, UPDATE és FETCH utasítások INTO részében használható. Példák:
```
SELECT … BULK COLLECT INTO kollekció FROM tábla WHERE …;
UPDATE tábla SET … RETURNING kifejezés BULK COLLECT INTO kollekció;
```

```
DELETE … RETURNING kifejezés BULK COLLECT INTO kollekció;
INSERT … RETURNING kifejezés BULK COLLECT INTO kollekció;
FETCH … BULK COLLECT INTO kollekció LIMIT n;
```

A PL/SQL nyelv használata SQL*PLUS környezetben:

PL/SQL procedúrákat a következőképpen hívhatunk meg SQL*PLUS-ból:

```
EXECUTE proc(param); -- vagy CALL proc(param);
```

A fenti mód ekvivalens azzal, mintha a következő pl/sql blokkot írnánk be:

```
BEGIN
  proc(param);
END;
```

SQL*PLUS-ban definiálhatunk úgynevezett session változót, ami a session végéig él. Ezt használhatjuk pl/sql blokkban is, úgy, mintha az egy host változó lenne (:változó). Pl. egy függvény által visszaadott értéket tehetünk bele, lehet egy procedúra IN OUT paramétere ... stb. Végül az aktuális értékét kiírhatjuk a képernyőre (vagy fájlba -> SPOOL)

Létrehozása: `VARIABLE v_név típus`

Kiírása: `PRINT v_név`

Használata különböző helyzetekben:

```
EXECUTE :v_név := érték;      (BEGIN  :v_név := érték;  END;)
EXECUTE :v_név := fv(param); (BEGIN :v_név := fv(param); END;)
EXECUTE proc(:v_név)          (BEGIN  proc(:v_név;  END;)
```

Függvényhívás szintaxisa:

Séma.Package.Fv_név@Db_link(paraméterek)

Megszorítások fv-ekre:

Ahhoz, hogy egy fv-t SQL utasításban lehessen használni a következő kritériumokat kell teljesítenie:

Tárolt fv legyen
Az argumentumai csak egyszerű típusúak lehetnek (nem lehet pl. oszlop)
Az összes formális paramétere IN módú legyen
Az összes formális paramétere belső Oracle adattípusú legyen
A visszatérési értéke belső Oracle típusú legyen

Mellékhatások fv-ekben:

Mellékhatást okozhat ha egy fv adatbázis táblára vagy package változóra hivatkozik. Az ilyen függvényeket nem használhatjuk tetszőleges SQL utasításban. Pl.

```
create or replace function rossz_fv return number is
begin
  INSERT INTO emp(ename) VALUES('kiss');
  RETURN 11;
end;

select rossz_fv from dual;
```

06571: Function ROSSZ_FV does not guarantee not to update database

Vagyis végrehajtáskor hibaüzenetet kapunk.

A tárolt fv-ek esetén az Oracle ellenőrizni tudja, hogy milyen mellékhatásai lehetnek a fv-nek és ennek megfelelően engedi meg a függvény használatát különböző esetekben. A package-beli

függvények viszont rejtve vannak így ezekre nekünk kell közölni a rendszerrel, hogy milyen mellékhatásai lehetnek a fv-nek. Ezt a package specifikációban egy PRAGMA-val tesszük meg, ami a deklaráció után kell hogy szerepeljen.
PRAGMA RESTRICT_REFERENCES(fv_név, WNDS [,WNPS] [,RNDS] [,RNPS]);
Ahol WNDS: writes no database state (nem módosítja az adatbázist)
WNPS: writes no package state (nem módosítja package változók értékét)
RNDS: reads no database state (nem kérdez le táblát)
RNPS: reads no package state (nem hivatkozik package változókra)
Az első megadása kötelező, a többi opcionális. Ezzel mondjuk meg az Oracle-nek, hogy a fv milyen mellékhatásokkal rendelkezhet (mennyire „tiszta" a fv) és az Oracle ez alapján dönti el, hogy milyen környezetekben fogja engedni a fv használatát.
Ha be akarjuk csapni és "szebbnek" mondjuk a fv-t mint amilyen azt fordításkor észreveszi a fordító és szól:

0/0 PL/SQL: Compilation unit analysis terminated
2/3 PLS-00452: Subprogram 'ROSSZ_FV' violates its associated pragma

Triggerek
A triggert SQL utasítással hozhatjuk létre (CREATE TRIGGER), de a trigger végrehajtható részét PL/SQL nyelven kell megírnunk. A trigger valamilyen esemény hatására automatikusan elindul és végrehajtja a PL/SQL blokkban megadott utasításokat. Ezen utasítások végrehajtásához a trigger tulajdonosának kell, hogy joga legyen, méghozzá közvetlenül és nem role-okon keresztül. Az esemény lehet DML utasítás (pl. insert, update), DDL utasítás (pl. create, drop), vagy adatbázis esemény (pl. startup, login).

DML triggerek

```
CREATE [OR REPLACE] TRIGGER [schema.]trigger
    {BEFORE | AFTER | INSTEAD OF}
    {DELETE | INSERT | UPDATE [OF column [, column] ...]}
[OR {DELETE | INSERT | UPDATE [OF column [, column] ...]}] ...
    ON [schema.]table
    [ [REFERENCING { OLD [AS] old [NEW [AS] new]
                   | NEW [AS] new [OLD [AS] old] } ]
    [ FOR EACH ROW
     [WHEN (condition)] ]
    pl/sql_block
```

A triggerhez tartozik egy **kiváltó (elsütő) művelet** (INSERT, DELETE, UPDATE).
A trigger egy **objektumhoz** (tábla vagy esetleg nézet) kötődik.
Időzítés: A trigger egy módosító művelet előtt vagy után (vagy helyette) fut le.
Trigger **típusa**: Ha megadjuk a FOR EACH ROW opciót akkor a trigger minden sorra egyszer végrehajtódik. Az ilyen triggereket **sor-trigger**nek hívjuk. Ellenkező esetben csak utasításonként egyszer hajtódik végre a trigger. Ekkor a neve **utasítás-trigger**.
When feltétel csak sortriggerre adható meg. Ilyenkor a trigger csak azokra a sorokra fut le, amelyek kielégítik a feltételt.
Triggerek **engedélyezhetők** vagy letilthatók (ALTER TRIGGER)
Ha egy művelet több triggert is aktivizál akkor azok futási sorrendje nem garantált.
Triggeren belül nem adható ki tranzakciókezelő utasítás. COMMIT, ROLLBACK, SAVEPOINT

Az oszlopok régi és új értékére a PL/SQL blokkban úgy hivatkozhatunk mint host változókra. (kettőspont a változó előtt: **:NEW.oszlop**, **:OLD.oszlop**)
BEFORE triggerben az új értéket meg is változtathatjuk és ekkor ez kerül be majd az oszlopba. AFTER trigger esetén ezt nem tehetjük meg.
Egy AFTER trigger viszont már használhatja a ROWID-jét a sornak.

A triggerek aktivizálódási sorrendje:
1. BEFORE utasítás szintű triggerek
2. Minden egyes érintett sorra
   a) a BEFORE sor szintű triggerek
   b) maga a DML utasítás és az integritási feltételek ellenőrzése
   c) az AFTER sor szintű triggerek
3. AFTER utasítás szintű triggerek

A trigger futása alatt a rendszer egy READ konzisztens állapotát garantálja minden hivatkozott táblának, így a trigger nem látja a futása alatt történt változásokat.

Mire vigyázzunk triggerek megadásakor?

Amit deklaratív módon is meg tud oldani az Oracle arra ne írjunk triggert.
Ne hozzunk létre rekurzív triggereket. (Pl. egy AFTER UPDATE trigger ne adjon ki update utasítást.)
INSERT esetén csak a NEW értékeknek van értelme, a régiek NULL-ok.
DELETE esetén csak az OLD értékeknek van értelme, az újak NULL-ok.
A WHEN után még nem kell kettőspont az OLD és NEW elé, csak a blokkban.
Ha több művelet elsütheti a triggert akkor így dönthetjük el melyik volt a tényleges:
IF INSERTING ... IF UPDATING [('oszlop')]... IF DELETING ...
Update esetén még az oszlopot is megtudhatjuk.
Ha a trigger közben kivétel lép fel, amit nem kezeltek le akkor a trigger és az elsütő művelet is ROLLBACK-elve lesz. Így lehet pl. triggerből visszacsévélni az eredeti műveletet. Gyakran erre a célra a RAISE_APPLICATION_ERROR(hibakód, hibaüzenet) procedúrát használják, mert ekkor a kiváltó műveletet kiadó program kultúráltan lekezelheti a hibát.
DDL és tranzakció-kezelő utasítás nem lehet a triggerben.
Információk a triggerekről: DBA_TRIGGERS

Megszorítások a triggerek használatával kapcsolatban:
Hivatkozó tábla elsődleges és idegen kulcs oszlopát nem módosíthatja a trigger. (Hivatkozó tábla az, amelyik idegen kulcs hivatkozásban van a módosított táblával, amire épp sor szintű trigger fut.) Így az alábbi csak akkor működik ha nincs idegen kulcs definíció a két tábla között.
Ez a 9i verziótól már akkor is működik, ha van idegen kulcs.

```
CREATE OR REPLACE TRIGGER cascade_upd
AFTER UPDATE OF deptno ON dept
FOR EACH ROW
BEGIN
  UPDATE emp SET emp.deptno = :new.deptno
  WHERE  emp.deptno = :old.deptno;
END;
```

Változás alatt lévő táblát nem olvashat a trigger. (Változás alatti az a tábla, amelynek módosítása alatt épp sor szintű trigger fut.) Vagyis egy sor szintű trigger nem olvashat egy éppen módosítás alatt levő táblából.

DDL és Adatbázis triggerek
Kiváltó eseményük lehet csak egy adott sémára, vagy az egész adatbázisra vonatkozó művelet.
Lehetséges DDL események: `CREATE, DROP, RENAME, DDL …`
Lehetséges adatbázis események: `SERVERERROR, LOGON, STARTUP …`

DBA_TRIGGERS.base_object_type -> database, schema, table, view

További infók: Application Developer's Guide 9. fejezet

Alprogramok, triggerek karbantartása

Alprogramokkal kapcsolatos rendszerjogosultságok:
```
CREATE (ANY) PROCEDURE …
EXECUTE (ANY) PROCEDURE …
```

Alprogramokkal kapcsolatos katalógusok:
DBA_OBJECTS
- CREATED amikor létrehozták
- LAST_DDL utolsó módosítás
- TIMESTAMP utolsó fordítás
- STATUS VALID/INVALID

DBA_SOURCE -- ebből olvas az SQLPLUS DESCRIBE utasítása
DBA_ERRORS -- ebből olvas az SQLPLUS SHOW ERRORS utasítása
```
SELECT     line|| '/' ||position POS,text
FROM  user_errors
WHERE      name = 'proc_nev'
ORDER BY line;
```
DBA_TRIGGERS (típus, esemény, tábla, when feltétel, státusz, forrás)

Adatbázis-objektumok között meglévő függőségek
Pl. egy procedúra hivatkozik egy táblára, egy függvényre, egy nézetre.
Ha a hivatkozott objektum megváltozik, akkor a hivatkozó INVALID állapotba kerül, és a legközelebbi hivatkozáskor újra fordítja a rendszer. Van közvetlen függőség és közvetett függőség.

USER_DEPENDENCIES és DBA_DEPENDENCIES nézetek
```
SELECT name, type, referenced_name, referenced_type
FROM   user_dependencies
WHERE  referenced_name IN ('EMP' , 'NEW_EMP' );
```

A fentinél elegánsabb módon nézhetők meg a függőségek két nézetből (DEPTREE, IDEPTREE). Ezek létrehozása és megfelelő feltöltése -> UTLDTREE.SQL

PUBLIC_DEPENDENCY (obj, hivatkozott_obj) nézet

PL/SQL komponensek újrafordítása
1. Automatikusan, amikor futás közben a rendszer INVALID-nak találja
2. Manuálisan

```
ALTER PROCEDURE <név> COMPILE
ALTER FUNCTION  <név> COMPILE
ALTER PACKAGE   <név> COMPILE {PACKAGE|BODY}
ALTER TRIGGER   <név> COMPILE
```

Újrafordításkor először minden invalid objektumot újrafordít a rendszer, amitől az illető függ.

Feltételes fordítás (10.2-es verziótól)
Példák a dokumentációból.

```
BEGIN
$IF DBMS_DB_VERSION.VER_LE_10_1 $THEN
  $ERROR 'unsupported database release' $END
$ELSE
  DBMS_OUTPUT.PUT_LINE ('Release ' || DBMS_DB_VERSION.VERSION || '.' ||
                        DBMS_DB_VERSION.RELEASE || ' is supported.');
  COMMIT WRITE IMMEDIATE NOWAIT;
$END
END;
/

CREATE PROCEDURE circle_area(radius my_pkg.my_real) IS
  my_area my_pkg.my_real;
  my_datatype VARCHAR2(30);
BEGIN
  my_area := my_pkg.my_pi * radius;
  DBMS_OUTPUT.PUT_LINE('Radius: ' || TO_CHAR(radius)
                       || ' Area: ' || TO_CHAR(my_area) );
  $IF $$my_debug $THEN -- if my_debug is TRUE, run some debugging code
    SELECT DATA_TYPE INTO my_datatype FROM USER_ARGUMENTS
       WHERE OBJECT_NAME = 'CIRCLE_AREA' AND ARGUMENT_NAME = 'RADIUS';
     DBMS_OUTPUT.PUT_LINE('Datatype of the RADIUS argument is: ' ||
my_datatype);
  $END
END;
/

ALTER PROCEDURE circle_area COMPILE PLSQL_CCFLAGS = 'my_debug:TRUE'
   REUSE SETTINGS;
```