

Capitolul 9. Proceduri, funcții și pachete PL/SQL

Dacă precedentul capitol a fost unul de familiarizare cu principalele ingrediente ale limbajului de programare Oracle, a sosit momentul pentru atacul frontal al unei componente esențiale pentru aplicațiile profesionale cu baze de date – procedurile stocate. Blocurile de până acum au fost anonime; pot fi salvate doar ca fișiere ASCII, fiind compilate la fiecare lansare în execuție.

În continuare, vom folosi toate cunoștințele dobândite pentru a redacta și folosi proceduri, funcții și pachete, forme sub care, alături de declanșatoare, programele PL/SQL pot face parte din schema unei baze de date Oracle. Spre deosebire de *blocurile anonime*, *blocurile cu nume* (numite sau denumite) prezintă o secțiune suplimentară, antetul, în care se specifică numele, parametrii și alte clauze specifice.

9.1. Proceduri

O procedură PL/SQL nu se deosebește, în principiu, de “suratele” sale din alte limbaje de programare, în sensul că este o secvență de program desemnată a executa anumite operațiuni. Ca orice bloc PL/SQL, procedura are trei secțiuni, plus zona declarativă. I se pot “pasa” parametri (care sunt de intrare sau intrare/ieșire), iar aceasta poate modifica parametri (de ieșire sau intrare/ieșire). Informațiile despre fiecare procedură pot fi aflate consultând două tabele ale dicționarului bazei de date: USER_OBJECTS și USER_SOURCE. La acestea se mai adaugă USER_PROCEDURES pe care o vom discuta în alt paragraf.

9.1.1. Creare/înlocuire

O procedură este creată prin comanda `CREATE PROCEDURE`. După creare, următoarea lansare a comenzii ar duce la declanșarea unei erori cum că procedura deja există în schemă, așa că în majoritatea cazurilor este preferată varianta `CREATE OR REPLACE PROCEDURE` care elimină acest inconvenient. Având ca element de raportare capitolul precedent, revenim la banalul bloc anonim dedicat rezolvării ecuației de gradul II (paragraful 8.3). Unul dintre cele mai vizibile neajunsuri ale blocului din listing 8.3 ține de faptul că parametrii *a*, *b* și *c* trebuie inițializați în zona declarativă. Cu un minim de efort, putem transforma blocul în procedura din listing 9.1.

Listing 9.1. Blocul anonim din listing 8.3 transformat în procedură

```
/* Prima procedură: Rezolvarea ecuației de gradul II (ediție revăzută a blocului din listing 8.3)*/  
CREATE OR REPLACE PROCEDURE p_ec2
```

```

(
  a IN INTEGER, b IN INTEGER, c IN INTEGER
)
AS
  delta NUMBER(16,2) ;
  x1 NUMBER(16,6) ;
  x2 NUMBER(16,6) ;

BEGIN
  ...nici o schimbare față de listing 8.3
END;
/

```

Procedura se numește P_EC2, și are trei parametri de intrare (IN) care sunt cei trei parametri ai ecuației, a, b, și c. Cuvântul cheie DECLARE lipsește, zona declarativă urmând cuvântului AS (sau IS). Zona executabilă a rămas identică celei din blocul inițial. Blocul este salvat pe disc cu numele Listing09_01.SQL. Lansarea sa în execuție va duce la crearea în baza de date a procedurii P_EC, după cum indică și figura 9.1.

```

SQL> @F:\ORACLE_CARTE\CAP09_PL_SQL2\LISTING09_01

Procedure created.

SQL> |

```

Figura 9.1. Lansarea în execuție a LISTING09_01.SQL și crearea procedurii

Odată creată, o procedură poate fi apelată din orice alt bloc, inclusiv dintr-unul anonim – vezi figura 9.2. Apelul presupune specificarea, în zona executabilă, a numelui procedurii și a valorilor parametrilor de intrare ai funcției. Astfel, 24 va constitui valoarea parametrului a, 555 valoarea lui b, iar 67 valoarea lui c.

```

SQL> BEGIN
  2   p_ec2 (24, 555, 67) ;
  3   END ;
  4   /
x1=-23.003642,   x2=-.121358

PL/SQL procedure successfully completed.

SQL>

```

Figura 9.2. Apelul procedurii dintr-un bloc anonim

În SQL*Plus o procedură poate fi apelată și fără a fi nevoie de un alt bloc PL/SQL, prin comanda EXECUTE:

```
EXECUTE p_ec2 (24, 555, 67)
```

9.1.2. Parametrii unei proceduri

În corpul unei funcții, un parametru de intrare (de tip IN) nu poate fi modificat. Regula generală este că parametrii de intrare (IN) pot fi doar “citiți”, parametri de

ieșire (OUT) doar “scriși” (în Oracle 8i și 9i parametrii de ieșire pot fi și “citiți”), iar cei de intrare-ieșire (IN-OUT) folosiți în ambele ipostaze. Implicit, categoria unui parametru este IN. Specificarea tipului fiecărui parametru poate fi, precum în cazul variabilelor, indirectă. Procedura P_POPULARE_PONTAJE_LUNA din listing 9.2 primește la execuție doi parametri, anul și luna pentru care se dorește popularea. Cei doi parametri sunt de același tip ca atributele corespondente din tabela SALARII.

Listing 9.2. Procedură pentru popularea tabelii PONTAJE pe o lună

```
-- Blocul anonim din listing 8.8 transformat în procedură
CREATE OR REPLACE PROCEDURE p_populare_pontaje_luna
(
  an_ IN salarii.an%TYPE, luna_salarii.luna%TYPE
)
IS
  prima_zi DATE ; -- variabila care stochează data de 1 a lunii
  zi DATE ; -- variabila folosită la ciclare
BEGIN
  prima_zi := TO_DATE('01/'|| luna_ || '/'|| an_ , 'DD/MM/YYYY') ;
  zi := prima_zi ;

  /* bucla se repetă pentru fiecare zi a lunii */
  WHILE zi <= LAST_DAY(prima_zi) LOOP
    IF RTRIM(TO_CHAR(zi,'DAY')) IN ('SATURDAY', 'SUNDAY') THEN -- e zi nelucrătoare
      NULL ;
    ELSE
      BEGIN -- de aici începe blocul inclus
        INSERT INTO pontaje (marca, data)
          SELECT marca, zi FROM personal ;
      EXCEPTION -- se preia eventuala violare a cheii primare
        WHEN DUP_VAL_ON_INDEX THEN
          -- se șterg mai întâi înregistrările pentru ziua curentă
          DELETE FROM pontaje WHERE data = zi ;
          -- apoi se reinserează înregistrările
          INSERT INTO pontaje (marca, data)
            SELECT marca, zi FROM personal ;
        END ; -- aici se termină blocul inclus
      END IF ;
      -- se trece la ziua următoare
      zi := zi + 1 ;
    END LOOP ;
    COMMIT ;
  END p_populare_pontaje_luna ;
/
```

Fie și numai pentru o primă exemplificare a apelului unei proceduri din altă procedură, listing-ul 9.3 generalizează popularea tabelii PONTAJE pentru un an specificat. La apelul P_POPULARE_PONTAJE_AN, valorile anul și i reprezintă parametrii *actuali* ai procedurii, în timp an_ și luna_ sunt parametrii *formali*. Parametrii actuali sunt, deci, valorile efective “pasate” procedurii în momentul apelului și, în cazul existenței unor parametri OUT sau IN OUT, rezultatele furnizate de procedură, în timp ce parametrii formali sunt cei care recepționează valorile parametrilor actuali, fiind folosiți conform logicii procedurii.

Listing 9.3. Procedură pentru popularea tabelii PONTAJE pe un an dat

```

-- Procedura pentru popularea tabelii PONTAJE pe un an întreg
CREATE OR REPLACE PROCEDURE p_populare_pontaje_an
( anul salarii.an%TYPE )
IS
BEGIN
  FOR i IN 1..12 LOOP
    DBMS_OUTPUT.PUT_LINE ('Urmeaza luna '|| i) ;
    p_populare_pontaje_luna (anul, i) ;
  END LOOP ;
END p_populare_pontaje_an ;

```

9.1.3. Exemplu de folosire a parametrilor de intrare/ieșire și recursivitate

Pornim de la un exemplu foarte simplu: crearea unei funcții care primește cinci parametri numerici pe care îi ordonează și îi afișează ordonați. Blocul din listing 9.4 este cel desemnat a întreprinde această operațiune.

Listing 9.4. Procedură pentru ordonarea a cinci numere – varianta 1

```

-- procedura de ordonare a 5 numere
CREATE OR REPLACE PROCEDURE ordonare_5
(n1 NUMBER, n2 NUMBER, n3 NUMBER, n4 NUMBER, n5 NUMBER )
IS
  -- cele cinci numere se preiau într-un tablou
  TYPE t_v IS TABLE OF NUMBER INDEX BY BINARY_INTEGER ;
  v t_v ;
  temp NUMBER (14,2) ; -- variabila folosită la schimbare
  o_schimbare BOOLEAN := TRUE ;
BEGIN
  v(1) := n1 ; -- inițializarea componentelor vectorului
  v(2) := n2 ;
  v(3) := n3 ;
  v(4) := n4 ;
  v(5) := n5 ;

  WHILE o_schimbare LOOP
    o_schimbare := FALSE ; -- presupunem că nu va exista nici o inversiune
    FOR i IN 1..4 LOOP
      IF v(i) > v(i+1) THEN /* ordinea nu e cea corectă, așa că se schimbă
                           între ele cele două componente alăturate */
        temp := v(i) ;
        v(i) := v(i+1) ;
        v(i+1) := temp ;
        o_schimbare := TRUE ;
      END IF ;
    END LOOP ;
  END LOOP ;
  DBMS_OUTPUT.PUT_LINE ('Ordinea finala : ' || v(1) || ' - ' || v(2) || ' - ' || v(3)
    || ' - ' || v(4) || ' - ' || v(5)) ;
END ;

```

Ideea procedurii este mai mult decât simplă: cele cinci numere sunt parametri de intrare ai procedurii; un tablou asociativ preia cele cinci valori pe care le compară, fiecare (exceptând pe ultima) cu următoarea. Ori de câte ori ordinea este greșită, se schimbă între ele valorile celor două componente. Procedura se repetă până când toate componentele sunt dispuse în ordine crescătoare. Pentru ciclare, variabila folosită este `o_schimbare`. Figura 9.3 ilustrează un mod de apelare dintr-un bloc anonim a procedurii. Cele cinci valori transmise procedurii constituie parametri de intrare care vor fi preluați în tabloul asociativ.

```
SQL> BEGIN
2  ORDONARE_5 (12,123,120,78,2);
3  END;
4  /
Ordinea finala :2 - 12 - 78 - 120 - 123

PL/SQL procedure successfully completed.
```

Figura 9.3. Apelul procedurii de ordonare

Este drept că acesta reprezintă unul dintre cei mai slabi algoritmi de sortare din cei pe care îi cunoaștem. Chiar și așa, am mai fi putut salva aparențele, reluând procesul de ordonare de la ultima poziție pe care s-a făcut schimbul.

Cu parametri de ieșire sau intrare-ieșire lucrurile stau altfel. Valorile acestora nu pot fi constante, deoarece blocul apelant trebuie să preia valorile obținute în urma prelucrărilor. Să complicăm lucrurile cu o a doua versiune a procedurii – cea din listing 9.5.

Listing 9.5. Procedură pentru ordonarea a cinci numere - varianta 2

```
-- procedură de ordonare a 5 numere - versiunea 2
CREATE OR REPLACE PROCEDURE ordonare_5v2 (
    nivel IN OUT PLS_INTEGER,
    n1 IN OUT NUMBER, n2 IN OUT NUMBER, n3 IN OUT NUMBER,
    n4 IN OUT NUMBER, n5 IN OUT NUMBER )
IS
    -- cele cinci numere se preiau într-un tablou
    TYPE t_v IS TABLE OF NUMBER INDEX BY BINARY_INTEGER ;
    v t_v ;
    temp NUMBER (14,2) ; -- variabila de lucru
    o_schimbare BOOLEAN := FALSE ;
BEGIN
    v(1) := n1 ; -- inițializarea componentelor vectorului
    v(2) := n2 ;
    v(3) := n3 ;
    v(4) := n4 ;
    v(5) := n5 ;

    /* se compară fiecare din primele 4 componente ale vectorului cu următoarea */
    FOR i IN 1..4 LOOP
        IF v(i) > v(i+1) THEN /* ordinea nu e cea corectă, așa că se schimbă
                               între ele cele două componente alăturate */
            temp := v(i) ;
            v(i) := v(i+1) ;
            v(i+1) := temp ;
            o_schimbare := TRUE ;
        END IF ;
    END LOOP ;
END ;
```

```

        DBMS_OUTPUT.PUT_LINE ('Nivel '||nivel||' - rezultat : ' || v(1) ||' - '||
            v(2) ||' - '|| v(3) ||' - '|| v(4) ||' - '||v(5));
        nivel := nivel + 1 ;
        -- apelul recursiv
        ordonare_5v2 (nivel, v(1), v(2), v(3), v(4), v(5)) ;
        nivel := nivel - 1 ;
        -- se modifică valorilor parametrilor IN OUT propriu-zis
        n1 := v(1) ;
        n2 := v(2) ;
        n3 := v(3) ;
        n4 := v(4) ;
        n5 := v(5) ;
        EXIT ;
    END IF ;
END LOOP ;
IF o_schimbare AND nivel < 1 THEN
    DBMS_OUTPUT.PUT_LINE ('Ordinea finala : ' || v(1) ||' - '|| v(2) ||' - '|| v(3)
        ||' - '|| v(4) ||' - '||v(5));
END IF ;
END ordonare_5v2 ;

```

Apelul dintr-un bloc anonim, în maniera primei versiuni, se va solda cu eșec – vezi figura 9.4.

```

SQL> BEGIN
2   ORDONARE_5v2 (0,12,123,120,78,2);
3   END;
4   /
ORDONARE_5v2 (0,12,123,120,78,2);
*
ERROR at line 2:
ORA-06550: line 2, column 16:
PLS-00363: expression '0' cannot be used as an assignment target
ORA-06550: line 2, column 18:
PLS-00363: expression '12' cannot be used as an assignment target
ORA-06550: line 2, column 21:
PLS-00363: expression '123' cannot be used as an assignment target
ORA-06550: line 2, column 25:
PLS-00363: expression '120' cannot be used as an assignment target
ORA-06550: line 2, column 29:
PLS-00363: expression '78' cannot be used as an assignment target
ORA-06550: line 2, column 32:
PLS-00363: expression '2' cannot be used as an assignment target
ORA-06550: line 2, column 2:
PL/SQL: Statement ignored

SQL>

```

Figura 9.4. Apelul eronat al unei proceduri cu parametri de intrare-ieșire (IN OUT)

Blocul anonim din listing 9.6 este cel care folosește ca parametri actuali ai procedurii variabilele nr1, nr2... Ordinea inițială este afișată în cadrul blocului, iar ordinea finală va fi listată de procedură. Pe lângă cele patru variabile ale căror valori trebuie ordonate, de data aceasta folosim o variabilă ce indică nivelul de auto-apel al procedurii - ordonare_5v2.

Listing 9.6. Bloc PL/SQL pentru apelul procedurii de ordonare

```

DECLARE
  /* folosim și variabila NIVEL pentru a pune în evidență apelul recursiv al procedurii */
  nivel PLS_INTEGER := 0 ;

  -- cele cinci valori de ordonat
  nr1 NUMBER(14,2) := 12 ;
  nr2 NUMBER(14,2) := 123 ;
  nr3 NUMBER(14,2) := 120 ;
  nr4 NUMBER(14,2) := 78 ;
  nr5 NUMBER(14,2) := 2 ;
BEGIN
  DBMS_OUTPUT.PUT_LINE ('Ordinea initiala : ' || nr1 || ' - ' || nr2 || ' - ' || nr3 || ' - ' || nr4 || ' - ' || nr5) ;
  ordonare_5v2 (nivel, nr1,nr2,nr3,nr4,nr5) ;
END ;

```

Rezultatul efectiv al execuției blocului anonim din listing 9.6 și urmărirea modului efectiv în care funcționează recursivitatea constituie subiectul figurii 9.5.

```

SQL> DECLARE
  2   nivel PLS_INTEGER := 0 ;
  3   nr1 NUMBER(14,2) := 12 ;
  4   nr2 NUMBER(14,2) := 123 ;
  5   nr3 NUMBER(14,2) := 120 ;
  6   nr4 NUMBER(14,2) := 78 ;
  7   nr5 NUMBER(14,2) := 2 ;
  8 BEGIN
  9   DBMS_OUTPUT.PUT_LINE ('Ordinea initiala : ' || nr1 || ' - ' || nr2 ||
10   ' - ' || nr3 || ' - ' || nr4 || ' - ' || nr5) ;
11   ordonare_5v2 (nivel, nr1,nr2,nr3,nr4,nr5) ;
12 END ;
13 /
Ordinea initiala :12 - 123 - 120 - 78 - 2
Nivel 0 - rezultat :12 - 120 - 123 - 78 - 2
Nivel 1 - rezultat :12 - 120 - 78 - 123 - 2
Nivel 2 - rezultat :12 - 78 - 120 - 123 - 2
Nivel 3 - rezultat :12 - 78 - 120 - 2 - 123
Nivel 4 - rezultat :12 - 78 - 2 - 120 - 123
Nivel 5 - rezultat :12 - 2 - 78 - 120 - 123
Nivel 6 - rezultat :2 - 12 - 78 - 120 - 123
Ordinea finala :2 - 12 - 78 - 120 - 123

PL/SQL procedure successfully completed.

SQL>

```

Figura 9.5. Apelul corect al procedurii cu parametri de intrare-ieșire (IN OUT)

9.1.4. Parametri OUT/IN OUT și excepții

Dacă într-o procedură sau funcție apare o excepție, sistem sau definită de utilizator, controlul este preluat de secțiunea `EXCEPTION` sau de blocul superior, iar parametrilor de tip `OUT` sau `IN OUT` nu li se transmite nici o valoare. Cu alte cuvinte, parametrii actuali (cei din procedura/funcția apelantă) vor aceeași valoare, ca și cum apelul n-ar fi avut loc. Pentru exemplificare, în cele ce urmează prezentăm o procedură în care se poate declanșa, la alegere, o excepție utilizator și

un bloc anonim care face apel la procedură în două variante, fără și cu declanșarea excepției. Listingul 9.7 conține procedura EROARE_CONTROLATĂ.

Listing 9.7. Procedura pentru declanșarea excepției

```
CREATE OR REPLACE PROCEDURE Eroare_Controlata
(se_declanseaza IN BOOLEAN,
parametru_de_IO IN OUT VARCHAR2)
AS
o_exceptie EXCEPTION ;
BEGIN
parametru_de_IO := 'Suntem la inceputul procedurii EROARE_CONTROLATA' ;
IF se_declanseaza THEN
RAISE o_exceptie ;
ELSE
parametru_de_IO := 'Parametru modificat' ;
END IF ;
END Eroare_Controlata ;
```

Prin parametrul de intrare `se_declanseaza` se controlează apariția erorii, de fapt, a excepției utilizatori (`o_exceptie`). Parametrul `de_IO` este, după cum indică numele, parametrul formal de intrare-ieșire a cărui valoare dorim să o observăm. La începutul secțiunii executabile parametrul `de_IO` primește valoarea 'Suntem la începutul procedurii EROARE_CONTROLATA'. Dacă la apelul procedurii, prin valoarea pasată parametrului `se_declanseaza`, execuția acesteia decurge fără generarea excepției, valoarea finală a parametrului de intrare-ieșire va fi 'Parametru modificat'. Blocul anonim din listing 9.8 apelează de două ori procedura, prima dată fără declanșarea excepției, a doua oară cu.

Listing 9.8. Blocul de test al valorilor parametrului actual

```
DECLARE
v_parametru_actual VARCHAR2(50) := 'Valoare inițială' ;
BEGIN
-- varianta 1 - FĂRĂ declanșarea excepției
Eroare_Controlata (FALSE, v_parametru_actual) ;
DBMS_OUTPUT.PUT_LINE ('v_parametru_actual = ' || v_parametru_actual) ;

-- varianta 2 - CU declanșarea excepției
v_parametru_actual := 'Valoare - VARIANTA 2' ;
Eroare_Controlata (TRUE, v_parametru_actual) ;
DBMS_OUTPUT.PUT_LINE ('v_parametru_actual = ' || v_parametru_actual) ;
EXCEPTION
WHEN OTHERS THEN
-- se preia eroarea din procedura EROARE_CONTROLATA
DBMS_OUTPUT.PUT_LINE ('v_parametru_actual in secțiunea EXCEPTION= '
|| v_parametru_actual) ;
END ;
```

Comportamentul parametrului de intrare-ieșire reiese destul de limpede lansând din SQL*Plus blocul anonim de mai sus – vezi figura 9.6.

```

SQL> DECLARE
2   v_parametru_actual VARCHAR2(50) := 'Valoare initiala' ;
3   BEGIN
4   -- varianta 1 - FARA declansarea exceptiei
5   Eroare_Controlata (FALSE, v_parametru_actual) ;
6   DBMS_OUTPUT.PUT_LINE ('v_parametru_actual = ' || v_parametru_actual) ;
7
8   -- varianta 2 - CU declansarea exceptiei
9   v_parametru_actual := 'Valoare - VARIANTA 2' ;
10  Eroare_Controlata (TRUE, v_parametru_actual) ;
11  DBMS_OUTPUT.PUT_LINE ('v_parametru_actual = ' || v_parametru_actual) ;
12
13  EXCEPTION
14  WHEN OTHERS THEN
15  -- se preia eroarea din procedura EROARE_CONTROLATA
16  DBMS_OUTPUT.PUT_LINE ('v_parametru_actual in sectiunea EXCEPTION= '
17  || v_parametru_actual) ;
18  END ;
19  /
v_parametru_actual = Parametru modificat
v_parametru_actual in sectiunea EXCEPTION= Valoare - VARIANTA 2

PL/SQL procedure successfully completed.

SQL>

```

Figura 9.6. Valoarea parametrului formal cu și fără excepție

Prima valoare afișată este cea stabilită în procedură. Întrucât al doilea apel al procedurii declanșează excepția, valoarea afișată este cea dinaintea apelului procedurii.

9.1.5. Opțiunea NOCOPY

Parametrii unei proceduri/funcții pot fi transmiși în două moduri, prin referință sau prin valoare. În primul caz, un pointer către parametrul actual este “pasat” parametrului formal corespondent. La transmiterea prin valoare are loc copierea valorii parametrului actual în cel formal. Prima variantă este mai rapidă, deoarece evită copierea, copiere care, atunci când parametrul este de tip colecție, poate consuma resurse importante.

Până în Oracle 8i, parametrii de intrare (IN) erau transmiși exclusiv prin referință, iar ceilalți prin valoare. Din Oracle 8i apare opțiunea NOCOPY la care programatorul poate apela în momentul declarării parametrului formal. Folosirea acestei opțiuni forțează compilatorul PL/SQL să folosească transmiterea prin referință. Când un parametru este transmis prin referință, orice modificare a parametrului formal determină modificarea parametrului actual, deoarece ambii “puntează” către aceeași locație. În consecință, dacă într-o procedură se declanșează o excepție nepreluată după momentul modificării parametrului formal, valoarea originală a parametrului actual se pierde.

După modelul lui Scott Urman¹ construim o procedură similară EROARE_CONTROLATA (listing 9.7), procedură numită P_NOCOPY și prezentată în listing 9.9, în care parametrul de intrare ieșire este acum pasat prin referință, și nu prin valoare (copiere).

Listing 9.9. Parametru IN OUT cu opțiunea NOCOPY

```
CREATE OR REPLACE PROCEDURE p_No_Copy
(se_declanseaza IN BOOLEAN, parametru_de_IO IN OUT NOCOPY VARCHAR2)
AS
o_exceptie EXCEPTION ;
BEGIN
parametru_de_IO := 'Valoare la inceputul procedurii P_NO_COPY' ;
IF se_declanseaza THEN
RAISE o_exceptie ;
ELSE
parametru_de_IO := 'Valoare modificata in sectiunea executabila a P_NO_COPY' ;
END IF ;
END p_No_Copy ;
```

Pentru a pune în valoare logica opțiunii NOCOPY recurgem la un bloc similar celui din listing 9.8, bloc în care procedura P_NO_COPY se apelează de două ori, prima dată fără declanșarea erorii, a doua oară cu declanșarea erorii, în ambele variante valorile parametrului actual înainte și după apel fiind cele din figura 9.7.

Listing 9.10. Valorile parametrului actual în condițiile folosirii opțiunii NOCOPY

```
DECLARE
v_parametru_actual VARCHAR2(60) := 'Valoare initiala in blocul anonim' ;
BEGIN
-- varianta 1 - FĂRĂ declanșarea excepției
p_No_Copy (FALSE, v_parametru_actual) ;
DBMS_OUTPUT.PUT_LINE ('Varianta 1 - dupa apel: '||v_parametru_actual) ;

-- varianta 2 - CU declanșarea excepției
v_parametru_actual := 'VARIANTA 2' ;
p_No_Copy (TRUE, v_parametru_actual) ;
DBMS_OUTPUT.PUT_LINE ('Varianta 2 - dupa apelul procedurii: '||v_parametru_actual) ;
EXCEPTION
WHEN OTHERS THEN
-- se preia eroarea din procedura p_NO_COPY
DBMS_OUTPUT.PUT_LINE ('Sectiunea EXCEPTION= ' || v_parametru_actual) ;
END ;
```

Comparativ cu rezultatele din figura anterioară, la al doilea apel al procedurii P_NO_COPY, cel care declanșează eroarea, valoarea parametrului actual dinaintea apelului se pierde.

¹ Vezi [Urman02], pp.372-377

```

SQL> @f:\oracle_carte\cap09_pl_sql2\listing09_10.sql
18 /
Varianta 1 - dupa apel: Valoare modificata in sectiunea executabila a P_NO_COPY
Sectiunea EXCEPTION= Valoare la inceputul procedurii P_NO_COPY

PL/SQL procedure successfully completed.

SQL>

```

Figura 9.7. Parametru de intrare-ieșire cu opțiunea NOCOPY

Nefiind o directivă de compilare, ci mai degrabă o “sugestie” adresată compilatorului PL/SQL, în general NOCOPY nu generează erori, ci este ignorată în cazurile problematice. Iată câteva situații de acest gen: parametrul actual este un element al unui vector asociativ; parametrul actual prezintă o restricție de tip NOT NULL, sau este numeric și limitat ca lungime sau poziții fracționare etc. Pentru detalii, vezi lucrarea lui Scott Urman². Ca avantaj major al acestei opțiuni, reținem creșterea semnificativă a vitezei atunci când parametrii pasași la apelul unei proceduri/funcții sunt de tip colecție.

9.1.6. Valori implicite ale parametrilor

Obligativ, la apelul unei proceduri sau funcții numărul parametrilor actuali trebuie să fie egal cu cel al parametrilor formali. Există, însă, o modalitate de a eluda această cerință – declararea de valori implicite pentru parametri. Să discutăm un exemplu simplu. Listing 9.3 conține procedura P_POPULARE_PONTAJE care prezintă drept parametru formal anul, și care lansează pentru fiecare lună procedura de populare a tabeli PONTAJE pe o lună calendaristică. Aducem o modificare minoră procedurii: pentru parametru definim valoarea implicită 2003 – vezi listing 9.11.

Listing 9.11. Procedura P_POPULARE_PONTAJE_AN - varianta 2

```

-- Procedura pentru popularea tabeli PONTAJE pe un an intreg - varianta 2
CREATE OR REPLACE PROCEDURE p_populare_pontaje_an
  ( anul salarii.an%TYPE DEFAULT 2003)
IS
BEGIN
  FOR i IN 1..12 LOOP

    DBMS_OUTPUT.PUT_LINE ('Urmeaza luna '|| i);
    p_populare_pontaje_luna (anul, i);

  END LOOP;

END p_populare_pontaje_an;

```

² [Urman02], p.374

În virtutea acestei modificări, procedura poate fi apelată fără a i se inițializa parametrul actual în zona declarativă sau executabilă. Figura 9.8 ilustrează lansarea în SQL*Plus a procedurii, cu ajutorul comenzii EXECUTE. Mesajele apărute pe ecran confirmă execuția procedurii de populare lunară.

```
SQL> EXECUTE P_POPULARE_PONTAJE_AN
Urmeaza luna 1
Urmeaza luna 2
Urmeaza luna 3
Urmeaza luna 4
Urmeaza luna 5
Urmeaza luna 6
Urmeaza luna 7
Urmeaza luna 8
Urmeaza luna 9
Urmeaza luna 10
Urmeaza luna 11
Urmeaza luna 12

PL/SQL procedure successfully completed.

SQL> |
```

Figura 9.8. Apel de procedură fără transmitere de parametri

9.1.7. Informații despre proceduri în dicționarul de date

Una din cele mai generoase tabele virtuale ale dicționarului de date este USER_OBJECTS din care, firește, nu puteau lipsi informațiile despre proceduri:

```
SELECT object_name, object_id, created,
       last_ddl_time, status
FROM user_objects
WHERE object_type = 'PROCEDURE' ;
```

Rezultatul acestei interogări - vezi figura 9.9. - furnizează: numele procedurii, codul unic al său în cadrul bazei, data creării, data ultimei modificări/compilări, precum și dacă este corectă sau are erori depistate la ultima compilare.

```

SQL> SELECT object_name, object_id, created, last_ddl_time, status
2 FROM user_objects
3 WHERE object_type = 'PROCEDURE' ;

```

OBJECT_NAME	OBJECT_ID	CREATED	LAST_DDL_	STATUS
ORDONARE_5	30837	16-FEB-03	16-FEB-03	VALID
ORDONARE_5V2	30840	16-FEB-03	16-FEB-03	VALID
P_EC2	30377	23-JAN-03	11-FEB-03	VALID
P_POPULARE_PONTAJE	30712	28-JAN-03	11-FEB-03	VALID
P_POPULARE_PONTAJE2	30713	28-JAN-03	18-FEB-03	VALID
P_POPULARE_PONTAJE_AN	30829	12-FEB-03	18-FEB-03	VALID
P_POPULARE_PONTAJE_LUNA	30828	12-FEB-03	17-FEB-03	VALID

```

7 rows selected.

SQL> |

```

Figura 9.9. Informații generale despre proceduri

Codul sursă al procedurilor, ca și al procedurilor și declanșatoarelor, se găsește în altă tabelă virtuală a catalogului de sistem – USER_SOURCE. Deși în SQL*Plus se poate recurge la comanda DESC (DESCRIBE), preferăm extragerea numelui și lungimii atributelor tabeli (virtuale) USER_SOURCE printr-o interogare aplicată dicționarului:

```

SELECT column_name, data_type, data_length
FROM all_tab_columns
WHERE table_name = 'USER_SOURCE'

```

Deoarece USER_SOURCE este o tabelă sistem, se interoghează nu USER_TAB_COLUMNS, ci ALL_TAB_COLUMNS.

```

SQL> SELECT column_name, data_type, data_length
2 FROM all_tab_columns
3 WHERE table_name = 'USER_SOURCE'
4 /

```

COLUMN_NAME	DATA_TYPE	DATA_LENGTH
NAME	VARCHAR2	30
TYPE	VARCHAR2	12
LINE	NUMBER	22
TEXT	VARCHAR2	4000

```

SQL>

```

Figura 9.10. Cele patru attribute ale tabeli

Figura 9.10 indică o structură pe cât de simplă, pe atât de utilă a tabeli:

- Name reprezintă numele procedurii;
- Type semnalizează dacă este vorba de o procedură, funcție, specificație de pachet, corp de pachet, declanșator sau tip definit de utilizator;

- `Line` este numărul liniei de cod;
- `Text` este chiar comanda de pe linia de program respectivă.

Astfel, pentru a afla corpul procedurii `P_POPULARE_PONTAJE_AN` fraza `SELECT` necesară este:

```
SELECT text
FROM user_source
WHERE name = 'P_POPULARE_PONTAJE_AN'
ORDER BY line
```

iar rezultatul este cel din figura 9.11. Clauza `ORDER BY line` a fost introdusă pentru a fi siguri că liniile procedurii vor fi dispuse corespunzător.

```
SQL> SELECT text
      2 FROM user_source
      3 WHERE name = 'P_POPULARE_PONTAJE_AN'
      4 ORDER BY line
      5 /

TEXT
-----
PROCEDURE P_POPULARE_PONTAJE_AN (
  anul salarii.an%TYPE)
IS
BEGIN
  FOR i IN 1..12 LOOP
    DBMS_OUTPUT.PUT_LINE ('Urmeaza luna '|| i) ;
    p_populare_pontaje_luna (anul, i) ;
  END LOOP ;
END ;

12 rows selected.

SQL>
```

Figura 9.11. Corpul procedurii `P_POPULARE_PONTAJE_AN`

Pe baza acestei opțiuni, în `SQL*Plus` putem crea un script care direcționează rezultatul interogării într-un fișier ASCII căruia o să-i spunem `f:\oracle_carte\p_populare_pontaje_an.sql`. Prin `SET HEADING OFF` se dezactivează antetele coloanelor. Cheia rezolvării problemei o constituie comanda `SPOOL` prin care rezultatul comenzilor ce-i succed vor fi stocate în fișierul ASCII cu numele și extensia specificate. `SPOOL OFF` oprește direcționarea rezultatelor în fișier.

Listing 9.12. Succesiuni de comezi `SQL*Plus` pentru salvarea într-un fișier ASCII a unei proceduri

```
SET HEADING OFF

SELECT text
FROM user_source
```

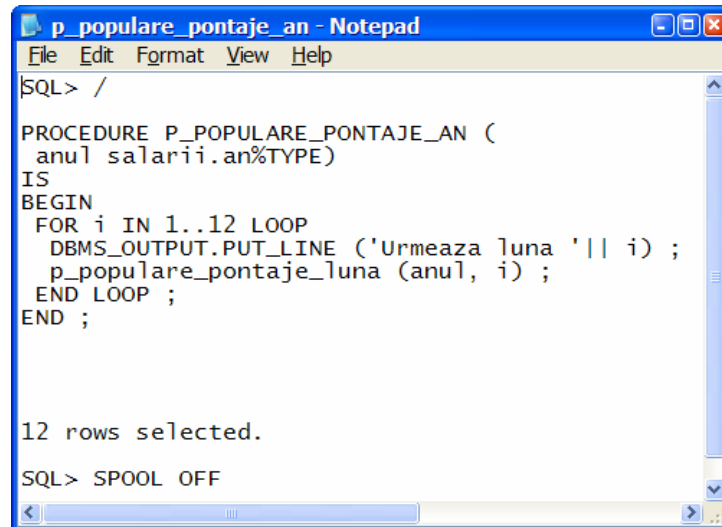
```

WHERE name = 'P_POPULARE_PONTAJE_AN'
ORDER BY line ;

SPOOL f:\oracle_carte\p_populare_pontaje_an.sql
/
SPOOL OFF
SET HEADING ON

```

Iată, în figura 9.12, conținutul fișierului ASCII creat. Este adevărat, pe lângă corpul propriu-zis al procedurii, mai apar și alte câteva linii nedorite, însă acestea nu diminuează decisiv farmecul soluției.



```

p_populare_pontaje_an - Notepad
File Edit Format View Help
SQL> /

PROCEDURE P_POPULARE_PONTAJE_AN (
  anul salarii.an%TYPE)
IS
BEGIN
  FOR i IN 1..12 LOOP
    DBMS_OUTPUT.PUT_LINE ('Urmeaza luna ' || i) ;
    p_populare_pontaje_luna (anul, i) ;
  END LOOP ;
END ;

12 rows selected.

SQL> SPOOL OFF

```

Figura 9.12. Fișier ASCII ce conține produra P_POPULARE_PONTAJE_AN

Folosind aceeași opțiune de direcționare a rezultatului unei fraze `SELECT` într-un fișier ASCII putem recompila toate procedurile din baza de date – vezi listing 9.13. Fraza `SELECT` are un artificiu: ordinea compilării este cea a identificatorului din cadrul bazei, identificator ce depinde de ordinea efectivă în care au fost create procedurile. Ideea este binevenită, deoarece se evită eventualele probleme ce ar putea să apară atunci când o procedură se compilează înaintea unei proceduri pe care o apelează.

Listing 9.13. Succesiuni de comezi SQL*Plus pentru recompilarea tuturor procedurilor

```

SET HEADING OFF
SELECT ' ALTER PROCEDURE ' || object_name || ' COMPILE ;'
FROM user_objects
WHERE object_type = 'PROCEDURE'
ORDER BY object_id ;
SPOOL f:\oracle_carte\cap09_PL_SQL2\recompilare_proceduri.txt
/
SPOOL OFF
SET HEADING ON
@f:\oracle_carte\cap09_PL_SQL2\recompilare_proceduri.txt

```

9.2. Funcții

De obicei, diferența dintre o funcție și o procedură este că procedura execută anumite operațiuni, în timp ce funcția întoarce o valoare. Diferența este una relativă, întrucât și procedurile și funcțiile pot avea doi sau mai mulți parametri de ieșire sau de intrare-ieșire care pot fi priviți ca rezultat al execuției blocului (subprogramului) din care fac parte. De asemenea, și funcțiile pot accepta valori implicite pentru parametri de intrare și opțiuni NOCOPY.

9.2.1. Creare/înlocuire

Similar procedurilor, o funcție este creată prin comanda CREATE FUNCTION care prezintă și forma CREATE OR REPLACE FUNCTION. Raportându-ne tot la celebrul exemplu dedicat ecuației de gradul II, să transformăm procedura din P_EC2 în funcție F_EC2. Valoarea returnată va fi șir de caractere, deoarece avem situații de nedeterminare, imposibilitate, rădăcini complexe și rădăcini complexe egale – vezi listing 9.14.

Listing 9.14. Funcția F_EC2

```

/* Prima funcție - Ecuația de gradul II se întoarce */
CREATE OR REPLACE FUNCTION f_ec2 (
  a IN INTEGER, b IN INTEGER, c IN INTEGER
) RETURN VARCHAR2
AS
  delta NUMBER(16,2);
  x1 NUMBER(16,6);
  x2 NUMBER(16,6);
  sir VARCHAR2(100) := '';
BEGIN
  -- ecuația este de gradul al II-lea ?
  IF a = 0 THEN
    IF b = 0 THEN
      IF c=0 THEN
        sir := 'Nedeterminare !';
      ELSE
        sir := 'Imposibil !!!';
      END IF;
    ELSE
      sir := 'Ecuația este de gradul I';
      x1 := -c / b;
      sir := sir || ', x=' || x1;
    END IF;
  ELSE
    delta := b**2 - 4*a*c;
    IF delta > 0 THEN
      x1 := (-b - SQRT(delta)) / (2 * a);
      x2 := (-b + SQRT(delta)) / (2 * a);
      sir := 'x1=' || x1 || ', x2=' || x2;
    ELSE
      IF delta = 0 THEN
        x1 := -b / (2 * a);
        sir := 'x1 = x2 = ' || x1;
      ELSE
        sir := 'Rădăcini complexe';
      END IF;
    END IF;
  END IF;
END

```



```

                                sir := 'Radacinile sunt complexe !!!' ;
                                END IF ;
                                END IF;
                                END IF ;
                                RETURN sir ;
                                END;

```

Comenzile de afișare din procedură au fost înlocuite cu cele de atribuire, scop în care s-a recurs la variabila `sir`, a cărei valoare este returnată la finalul funcției. Apelul funcției dintr-un bloc anonim este ilustrat în figura 9.13.

```

SQL> BEGIN
      2  DBMS_OUTPUT.PUT_LINE( F_EC2 (24, 555, 67) );
      3  END;
      4  /
x1=-23.003642, x2=-.121358

PL/SQL procedure successfully completed.

SQL>

```

Figura 9.13. Apelul funcției `F_EC2` dintr-un bloc anonim

Pentru baza de date legată de gestionarea salarizării, ne propunem să creăm o funcție simplă care să primească drept parametru de intrare marca și să furnizeze salariul orar al angajatului respectiv. Atunci când marca este eronată, adică nu există în `PERSONAL` nici un om al muncii cu marca respectivă, se dorește ca valoarea returnată să fie zero. Pentru aceasta, funcția folosește, după cum se observă în listing 9.15, un bloc inclus în cel principal, astfel încât excepția `NO_DATA_FOUND` să fie preluată.

Listing 9.15. Funcția `F_AFLA_SALORAR`

```

/* Funcție care întoarce salariul orar al unui angajat pe baza mărcii */
CREATE OR REPLACE FUNCTION f_afla_salorar
(
    marca_personal.marca%TYPE
) RETURN personal.salorar%TYPE
AS
    v_salorar personal.salorar%TYPE ;
BEGIN
    -- folosim un bloc inclus pentru a returna 0 când marca este eronată
    BEGIN
        SELECT salorar INTO v_salorar FROM personal WHERE marca = marca_ ;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN -- marca indicată nu există în PERSONAL
            v_salorar := 0 ;
    END ;
    RETURN v_salorar ;
END;
/

```

Până la a folosi această funcție, ne propunem să modificăm corpul procedurii `P_POPULARE_PONTAJE_LUNA` din listing 9.2. Ideea este de a evita declanșarea excepției `DUP_VAL_ON_INDEX`, prin apelul la o funcție care verifică dacă pentru

angajatul și ziua curente există deja o înregistrare în PONTAJE. Listing 9.16 prezintă corpul acestei funcții – F_ESTE_IN_PONTAJE.

Listing 9.16. Funcția F_ESTE_IN_PONTAJE

```

/* Funcție care întoarce TRUE dacă există pontaj pentru marca și ziua curente */
CREATE OR REPLACE FUNCTION f_este_in_pontaje (
    marca_personal.marca%TYPE, data_pontaje.data%TYPE
) RETURN BOOLEAN
AS
    v_este BOOLEAN ;
    v_unu INTEGER ;
BEGIN
    -- folosim un bloc inclus
    BEGIN
        SELECT 1 INTO v_unu FROM pontaje WHERE marca=marca_ AND data=data_ ;
        v_este := TRUE ;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN -- nu există înregistrarea în PONTAJE
            v_este := FALSE ;
    END ;
    RETURN v_este ;
END;
/

```

Noua versiune a procedurii P_POPULARE_PONTAJE_LUNA nu mai introduce deodată, pentru o zi lucrătoare, înregistrări pentru toți angajații, ci folosește un cursor în care se încarcă mărcile tuturor angajaților. La trecerea prin cursor, se verifică, dacă pentru ziua zi și marca rec_marci.marca, există înregistrare corespundentă în PONTAJE. În caz că nu, se înserează o linie (vezi listing 9.17).

Listing 9.17. Procedură pentru popularea tabeli PONTAJE pe o lună

```

CREATE OR REPLACE PROCEDURE p_populare_pontaje_luna
    (an_ IN salarii.an%TYPE, luna_ salarii.luna%TYPE )
IS
    prima_zi DATE ; -- variabila care stocheaza data de 1 a lunii
    zi DATE ; -- variabila folosită la ciclare
BEGIN
    prima_zi := TO_DATE('01/'||luna_|| '/'||an_, 'DD/MM/YYYY') ;
    zi := prima_zi ;

    /* bucla se repetă pentru fiecare zi a lunii */
    WHILE zi <= LAST_DAY(prima_zi) LOOP
        IF RTRIM(TO_CHAR(zi,'DAY')) IN ('SATURDAY', 'SUNDAY') THEN
            -- e zi nelucrătoare (sâmbătă sau duminică)
            NULL ;
        ELSE
            FOR rec_marci IN (SELECT marca FROM personal) LOOP
                IF f_este_in_pontaje (rec_marci.marca, zi) THEN
                    NULL ; -- păstrăm înregistrarea existentă
                ELSE
                    INSERT INTO pontaje (marca, data)
                        VALUES (rec_marci.marca, zi) ;
                END IF ;
            END LOOP ;
        END LOOP ;
    END LOOP ;

```

```

        END IF ;
        -- se trece la ziua urmatoare
        zi := zi + 1 ;
    END LOOP ;
    COMMIT ;
END ;

```

9.2.2. Apelarea funcțiilor din alte blocuri

Firește, orice funcție poate fi apelată din altă procedură, funcție sau bloc anonim. Spre deosebire de proceduri, funcția poate fi inclusă într-o expresie specifică unei comenzi de atribuire sau test. Începem discuția cu funcția `F_ANI_VECHIME`, prezentată în listing 9.18, care primește valorile a trei parametri – data de la care se calculează sporul de vechime, anul și luna de referință – și, pe baza acestora, furnizează numărul anilor de vechime, de fapt, a anilor scurși între `datasv` și 01-luna_-an_ (data de întâi a lunii de referință).

Listing 9.18. Calculul numărului de ani dintre o dată și prima zi a lunii de referință

```

CREATE OR REPLACE FUNCTION f_an_i_vechime
(
    datasv_personal.datasv%TYPE,
    an_IN_salarii.an%TYPE,
    luna_salarii.luna%TYPE
) RETURN transe_sv.ani_limita_inf%TYPE
AS
    -- variabila care stochează data de 1 a lunii
    prima_zi DATE := TO_DATE('01/'||luna_||'/'||an_, 'DD/MM/YYYY') ;
BEGIN
    RETURN TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0) ;
END ;

```

Funcția nu ridică probleme majore de comprehensibilitate. Cu funcția-sistem `MONTHS_BETWEEN` se determină numărul lunilor dintre cele două date-reper, număr care se împarte la 12 pentru a afla anii. Deoarece interesează numai partea întreagă a raportului (numărul de ani *împliniți*), se folosește funcția sistem `TRUNC`.

A doua funcție – `F_PROCENT_SPOR_VECHIME` – primește ca parametru de intrare un număr de ani și returnează procentul sporului de vechime corespunzător. Informația se obține prin interogarea tabeli `TRANȘE_SV` – vezi listing 9.19.

Listing 9.19. Aflarea procentului de spor de vechime corespunzător unui număr de ani (tot de vechime)

```

CREATE OR REPLACE FUNCTION f_procent_spor_vechime
(
    ani_transe_sv.ani_limita_inf%TYPE
) RETURN transe_sv.procent_sv%TYPE
AS
    v_procent transe_sv.procent_sv%TYPE ;
BEGIN
    SELECT procent_sv INTO v_procent FROM transe_sv
        WHERE ani_ >= ani_limita_inf AND ani_ < ani_limita_sup ;
    RETURN v_procent ;
EXCEPTION

```

```

WHEN NO_DATA_FOUND THEN
    RETURN 0 ;
END ;

```

Dacă valoarea primită drept parametru nu se încadrează în nici un interval din `TRANSE_SV`, se va returna zero, considerându-se o exagerare.

Funcția `F_PERSONAL` din listing 9.20 e un pic mai pretențioasă, în sensul că returnează, la cerere, pentru un angajat valoarea unuia dintre atributele: `SalOrar`, `SalOrarCO` sau `DataSV`. De aceea există doi parametri de intrare, unul pentru marca angajatului care interesează și un altul care desemnează valoarea cărui atribut va fi furnizată.

Listing 9.20. Funcție care furnizează valoarea unui atribut la cerere

```

CREATE OR REPLACE FUNCTION f_personal
(marca_personal.marca%TYPE, atribut_ VARCHAR2)
RETURN VARCHAR2
AS
    v_valoare VARCHAR2(50) ;
BEGIN
    CASE
        WHEN UPPER(atribut_) = 'SALORAR' THEN
            SELECT TO_CHAR(salorar, '999999999999999') INTO v_valoare
            FROM personal WHERE marca=marca_ ;

        WHEN UPPER(atribut_) = 'SALORARCO' THEN
            SELECT TO_CHAR(salorarco, '999999999999999') INTO v_valoare
            FROM personal WHERE marca=marca_ ;

        WHEN UPPER(atribut_) = 'DATASV' THEN
            SELECT TO_CHAR(datasv, 'DD/MM/YYYY') INTO v_valoare
            FROM personal WHERE marca=marca_ ;
    END CASE ;

    RETURN v_valoare ;

EXCEPTION
WHEN NO_DATA_FOUND THEN
    RETURN NULL ;
END ;

```

Deoarece cele trei valori sunt eterogene (două numerice și una dată calendaristică), valoarea returnată este de tip șir de caractere de lungime variabilă, urmând că blocul apelant să opereze conversiile de rigoare. În plus, dacă nu există nici un angajat cu marca specificată, valoarea furnizată este `NULL`.

Funcția următoare, `F_EXISTA_SP_RE_SA` (de la *f_există în sporuri rețineri salarii* - vezi listing 9.21) are deopotrivă ceva din `F_PERSONAL` (listing 9.20) și `F_ESTE_IN_PONTAJE` (listing 9.16), în sensul că valoarea returnată este `TRUE` sau `FALSE`, însă căutarea existenței unei înregistrări pentru marca, anul și luna specificate se face, la alegere, într-una din tabelele `SPORURI`, `REȚINERI` și `SALARII`. Așa încât, la cei trei parametri deja menționați, se mai adaugă un al patrulea ce indică în ce tabelă are loc căutarea.

Listing 9.21. Funcție de căutare în trei tabele

```

CREATE OR REPLACE FUNCTION f_exista_sp_re_sa (
    marca_personal.marca%TYPE,
    an_salarii.an%TYPE,
    luna_salarii.luna%TYPE,
    tabela VARCHAR2
) RETURN BOOLEAN
AS
    v_unu NUMBER(1);
BEGIN
    CASE
        WHEN UPPER(tabela) = 'SPORURI' THEN
            SELECT 1 INTO v_unu FROM sporuri
            WHERE marca=marca_ AND an=an_ AND luna=luna_ ;

        WHEN UPPER(tabela) = 'RETINERI' THEN
            SELECT 1 INTO v_unu FROM retineri
            WHERE marca=marca_ AND an=an_ AND luna=luna_ ;

        WHEN UPPER(tabela) = 'SALARII' THEN
            SELECT 1 INTO v_unu FROM salarii
            WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
    END CASE ;

    RETURN TRUE ;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE ;
END ;

```

Rezultatul furnizat depinde de succesul comenzilor `SELECT`. Astfel, dacă se găsește (cel puțin) o înregistrare, se ajunge în corpul procedurii la comanda `RETURN TRUE`. Altminteri, se declanșează excepția `NO_DATA_FOUND`, așa încât în secțiunea de tratare a acesteia a fost inclus un `RETURN FALSE`.

Funcțiile create în acest paragraf ne vor ajuta la “schimbarea la față” la blocului de actualizare a tabelelor `SPORURI` și `SALARII` pe baza datelor din `PONTAJE` pentru o lună dată, pe care, dacă tot avem prilejul, îl și transformăm în procedură – `P_ACT_SP_SA` (*p_actualizare_sporuri_retineri_salarii*) – listing 9.22.

Listing 9.22. Procedura de actualizare a tabelelor `SPORURI` și `SALARII`

```

/* procedura de actualizare, a tabelelor SPORURI și SALARII pe baza datelor din PONTAJE */
CREATE OR REPLACE PROCEDURE p_act_sp_sa
    (an_salarii.an%TYPE, luna_salarii.luna%TYPE)
AS
    -- C_ORE calculează totalul orelor lucrate, de concediu și de noapte pentru luna dată
    CURSOR c_ore IS
        SELECT marca, SUM(orelucrate) AS ore_l, SUM(oreco) AS ore_co,
            SUM(orenoapte) AS ore_n
        FROM pontaje
        WHERE TO_NUMBER(TO_CHAR(data,'YYYY')) = an_
            AND TO_NUMBER(TO_CHAR(data,'MM')) = luna_
        GROUP BY marca ;

```

```

v_spvech sporuri.spvech%TYPE ;
v_venitbaza salarii.venitbaza%TYPE ;
v_sпноapte sporuri.spноapte%TYPE ;
v_sporuri salarii.sporuri%TYPE ;

BEGIN
  FOR rec_ore IN c_ore LOOP

    /* se calculează venitul de bază, sporul de vechime și sporul de noapte;
       funcția ROUND asigură rotunjirea la ordinul sutelor */
    v_venitbaza := ROUND( rec_ore.ore_l * NVL(TO_NUMBER (f_personal
(rec_ore.marca, 'SALORAR')),0) + rec_ore.ore_co *
NVL(TO_NUMBER(f_personal(rec_ore.marca, 'SALORARCO')),0),-2) ;

    v_spvech := ROUND(v_venitbaza * f_procent_spor_vechime( f_anii_vechime(
TO_DATE(f_personal(rec_ore.marca, 'DATASV') , 'DD/MM/YYYY'), an_, luna_)
/ 100, -3) ;

    v_sпноapte := ROUND(rec_ore.ore_n * TO_NUMBER(f_personal(
rec_ore.marca, 'SALORAR')) * .15, -3) ;

    IF f_exista_sp_re_sa (rec_ore.marca, an_, luna_, 'SPORURI') THEN
      -- se actualizează tabela SPORURI pentru angajatul curent
      UPDATE sporuri
      SET spvech = v_spvech, oreноapte = rec_ore.ore_n, spноapte = v_sпноapte
      WHERE marca=rec_ore.marca AND an=an_ AND luna=luna_ ;
    ELSE
      INSERT INTO sporuri VALUES (rec_ore.marca, an_, luna_,
v_spvech, rec_ore.ore_n, v_sпноapte, 0) ;
    END IF ;

    -- se procedează analog pentru tabela SALARII
    IF f_exista_sp_re_sa (rec_ore.marca, an_, luna_, 'SALARII') THEN
      UPDATE salarii
      SET orelucrate = rec_ore.ore_l, oreco = rec_ore.ore_co,
venitbaza = v_venitbaza,
sporuri = (SELECT spvech + spноapte + altesp
FROM sporuri WHERE an=an_ AND luna=luna_
AND marca = rec_ore.marca)
WHERE marca=rec_ore.marca AND an=an_ AND luna=luna_ ;
    ELSE
      INSERT INTO salarii VALUES (rec_ore.marca, an_, luna_, rec_ore.ore_l,
rec_ore.ore_co, v_venitbaza,
(SELECT spvech + spноapte + altesp
FROM sporuri WHERE an=an_ AND luna=luna_ AND
marca = rec_ore.marca), 0, 0) ;
    END IF ;
  END LOOP ;
  COMMIT ;
END p_act_sp_sa ;

```

De data aceasta, calculul venitului de bază, a sporului de vechime, sporului de noapte, precum și verificarea existenței înregistrărilor în SPORURI și SALARII se realizează cu concursul funcțiilor anterioare.

9.2.3. Funcții apelabile din interogări SQL

Vestea bună este că multe funcții utilizator pot fi apelate din fraze `SELECT` în mod similar funcțiilor sistem. Vestea proastă este că nu toate funcțiile au acest privilegiu. Una dintre poruncile majore este ca o funcție invocată într-un `SELECT` să nu modifice baza de date. Să luăm un prim exemplu. Pentru a afișa fiecărui angajat numărul de ani de vechime la data de 1 iunie 2003 putem recurge la funcția `F_ANI_VECHIME` astfel:

```
SELECT marca, numepren, datasv,
       f_an_i_vechime(datasv, 2003,6) AS Ani_Vechime
FROM PERSONAL
```

Situația obținută în `SQL*Plus` se prezintă ca în figura 9.14.

```
SQL> SELECT marca, numepren, datasv,
2      f_an_i_vechime(datasv, 2003,6) AS Ani_Vechime
3      FROM PERSONAL
4      /
```

MARCA	NUMEPREN	DATASV	ANI_VECHIME
101	Angajat 1	12-OCT-80	22
102	Angajat 2	12-NOV-78	24
103	Angajat 3	02-JUL-76	26
104	Angajat 4	05-JAN-82	21
105	Angajat 5	12-NOV-77	25
106	Angajat 6	11-APR-85	18
107	Angajat 7	12-NOV-71	31
108	Angajat 8	12-NOV-99	3
109	Angajat 9	05-JAN-82	21
110	Angajat 10	05-JAN-82	21
1009	Angajat Nou	11-FEB-03	0
1001	Angajat Si Mai Nou	11-FEB-03	0
1002	Angajat Si Mai Nou2	11-FEB-03	0
1003	Angajat Si Mai Nou3	11-FEB-03	0

```

14 rows selected.

SQL>
```

Figura 9.14. Apelul funcției `F_ANI_VECHIME` dintr-o frază `SELECT`

Dacă, alături de informațiile din figura 9.14, dorim obținerea și a procentului sporului de vechime, calculat pe baza intervalelor din tabela `TRANSE_SV`, facem apel la funcția `F_PROCENT_SPOR_VECHIME` căreia îi pasăm ca parametru actual rezultatul funcției `F_ANI_VECHIME`:

```
SQL> SELECT marca, numepren, datasv,
2   f_ani_vechime(datasv, 2003,6) AS Ani_Vechime,
3   f_procent_spor_vechime(
4   f_ani_vechime(datasv, 2003,6)
5   ) || ' %' AS Procent_SV
6 FROM PERSONAL
7 /
```

MARCA	NUMEPREN	DATASV	ANI_VECHIME	PROCENT_SV
101	Angajat 1	12-OCT-80	22	25 %
102	Angajat 2	12-NOV-78	24	25 %
103	Angajat 3	02-JUL-76	26	25 %
104	Angajat 4	05-JAN-82	21	25 %
105	Angajat 5	12-NOV-77	25	25 %
106	Angajat 6	11-APR-85	18	18 %
107	Angajat 7	12-NOV-71	31	25 %
108	Angajat 8	12-NOV-99	3	5 %
109	Angajat 9	05-JAN-82	21	25 %
110	Angajat 10	05-JAN-82	21	25 %
1009	Angajat Nou	11-FEB-03	0	0 %
1001	Angajat Si Mai Nou	11-FEB-03	0	0 %
1002	Angajat Si Mai Nou2	11-FEB-03	0	0 %
1003	Angajat Si Mai Nou3	11-FEB-03	0	0 %

14 rows selected.

SQL> |

Figura 9.15. Frază SELECT în care o funcție apelează o altă funcție

```
SELECT marca, numepren, datasv,
      f_ani_vechime(datasv, 2003,6) AS Ani_Vechime,
      f_procent_spor_vechime(f_ani_vechime(datasv, 2003,6)
      ) || ' %' AS Procent_SV
FROM PERSONAL
```

Rezultatul este cel din figura 9.15. Detalii despre restricțiile ce trebuie îndeplinite de funcțiile apelabile în fraze SELECT (nivelele de puritate, după expresia PL/SQL-iștilor) vă vor fi prezentate într-un alt paragraf, către finalul capitolului.

9.2.4. Informații despre funcții în dicționarul de date

Similar procedurilor, informațiile generale despre funcții se obțin USER_OBJECTS:

```
SELECT *
FROM user_objects
WHERE object_type = 'FUNCTION'
```

Codul sursă al oricărei funcții se află (tot) în tabela virtuală a dicționarului USER_SOURCE:

```
SELECT text
FROM user_source
WHERE name = 'F_EXISTA_SP_RE_SA'
ORDER BY line
```


După modelul prezentat în listing 9.7 în SQL*Plus se poate salva într-un fișier ASCII corpul unei funcții, iar în listing 9.23 se prezintă succesiunea de comenzi SQL*Plus pentru recompilarea tuturor funcțiilor.

Listing 9.23. Succesiuni de comenzi SQL*Plus pentru recompilarea tuturor funcțiilor

```
SET HEADING OFF
SELECT ' ALTER FUNCTION ' || object_name || ' COMPILE ;'
FROM user_objects
WHERE object_type = 'FUNCTION'
ORDER BY object_id ;
SPOOL f:\oracle_carte\cap09_PL_SQL2\recompilare_funcatii.txt
/
SPOOL OFF
SET HEADING ON
@f:\oracle_carte\cap09_PL_SQL2\recompilare_funcatii.txt
```

9.3. Pachete

Pachetele reprezintă unul dintre cele mai interesante ingrediente PL/SQL, preluat tot din limbajul Ada. Un pachet, după cum îi zice numele, grupează obiecte procedurale de tipul procedurilor, funcțiilor, variabilelor, excepțiilor, cursorilor și tipurilor. Lucrurile sunt cu atât mai captivante cu cât fiecare pachet este compus din două părți stocate separat în dicționarul de date, *specificațiile*, care sunt publice și în care sunt “anunțate” toate elementele de sub umbrela pachetului, și *corpul*, în care sunt descrise, efectiv, funcțiile și procedurile.

Interesul este cu atât mai mare cu cât toate variabilele și cursorii declarate în specificații sunt publice (globale) și pot fi accesate/manipulate din orice alt bloc, anonim sau nu, pe toată durata sesiunii de lucru. Altfel spus, clasicele variabile publice/globale din alte limbaje de programare sunt disponibile în PL/SQL sunt forma unor variabile declarate în specificațiile unui pachet.

9.3.1. Specificații

În cele ce urmează ne propunem să creăm un pachet în care să regroupăm o serie de proceduri și funcții deja discutate, sau versiuni ameliorate ale acestora. Listing-ul 9.24 conține partea publică, de specificații a pachetului. Un mare avantaj ține de faptul că ordinea declarării nu este atât de rigidă, prin comparație cu gestionarea “independentă” a procedurilor și funcțiilor. Atunci când parametrii unei funcții/proceduri/cursor fac referință la o variabilă, aceasta trebuie să fi fost deja declarată în momentul referinței.

Listing 9.24. Specificațiile pachetului PACHET_SALARIZARE

```
CREATE OR REPLACE PACKAGE pachet_salarizare AS

/* primele două variabile globale, AN_ și LUNA_ preiau valorile
   inițiale din data sistemului */
```

```

v_an salarii.an%TYPE := TO_NUMBER(TO_CHAR(SYSDATE, 'YYYY'));
v_luna salarii.luna%TYPE := TO_NUMBER(TO_CHAR(SYSDATE, 'MM'));

-- variabila pentru păstrarea liniilor din TRANSE_SV
TYPE t_transe_sv IS TABLE OF transe_sv%ROWTYPE INDEX BY PLS_INTEGER;
v_transe_sv t_transe_sv;

-- procedura de inițializare a vectorului asociativ V_TRANSE_SV
PROCEDURE p_init_v_transe_sv;

/* se declară un vector asociativ pentru stocarea salariilor orare,
salariilor orare pentru calculul indemnizației de concediu și sporului
de vechime; indexul este chiar marca */
TYPE r_personal IS RECORD (
    salorar personal.salorar%TYPE, salorarCO personal.salorarCO%TYPE,
    datasv personal.datasv%TYPE);

TYPE t_personal IS TABLE OF r_personal INDEX BY PLS_INTEGER;
v_personal t_personal;

PROCEDURE p_init_vectori_personal;

/* mutăm în pachet o serie de funcții create anterior */
FUNCTION f_este_in_pontaje (marca_ personal.marca%TYPE,
    data_ pontaje.data%TYPE) RETURN BOOLEAN;

FUNCTION f_este_in_sporuri (marca_ personal.marca%TYPE,
    an_ v_an%TYPE, luna_ v_luna%TYPE) RETURN BOOLEAN;

FUNCTION f_este_in_retineri (marca_ personal.marca%TYPE,
    an_ v_an%TYPE, luna_ v_luna%TYPE) RETURN BOOLEAN;

FUNCTION f_este_in_salarii (marca_ personal.marca%TYPE,
    an_ v_an%TYPE, luna_ v_luna%TYPE) RETURN BOOLEAN;

FUNCTION f_anii_vechime (datasv_ personal.datasv%TYPE,
    an_ IN salarii.an%TYPE, luna_ salarii.luna%TYPE
    ) RETURN transe_sv.ani_limita_inf%TYPE;

FUNCTION f_procent_spor_vechime (ani_ transe_sv.ani_limita_inf%TYPE)
    RETURN transe_sv.procent_sv%TYPE;

CURSOR c_ore (an_ v_an%TYPE, luna_ v_luna%TYPE) IS
    SELECT marca, SUM(orelucreate) AS ore_l, SUM(oreco) AS ore_co,
        SUM(orenoapte) AS ore_n
    FROM pontaje
    WHERE TO_NUMBER(TO_CHAR(data,'YYYY')) = an_ AND
        TO_NUMBER(TO_CHAR(data,'MM')) = luna_ GROUP BY marca;

prea_multe_ore EXCEPTION;

END pachet_salarizare;

```

Pachetul conține aproape de toate:

- variabile globale scalare: v_an, v_luna;
- un tip înregistrare: r_personal;
- tipuri colecție: t_transe_sv, t_personal;

- variabile globale de tip colecție: `v_transe_sv`, `v_personal`;
- proceduri: `P_INIT_TRANSE_SV`, `P_INIT_VECTORI_PERSONAL`;
- funcții: `F_ESTE_IN_PONTAJE`, `F_ESTE_IN_SPORURI`, `F_ESTE_IN_RETINERI`, `F_ESTE_IN_SALARII`, `F_ANI_VECHIME`, `F_PROCENT_SPOR_VECHIME`;
- un cursor: `C_ORE`;
- excepție: `prea_multe_ore`.

9.3.2. Corpul pachetului

După lansarea în execuție a scriptului ce conține specificațiile, se trece la descrierea efectivă a procedurilor și funcțiilor enumerate în specificații, altfel spus, se crează corpul pachetului. Variabilele, cursoarele, tipurile și excepțiile declarate în corpul pachetului vor avea caracter local, nefiind accesibile decât în blocurile în care au fost definite.

Corpul pachetului, în care sunt incluse, în întregime, procedurile și funcțiile declarate în specificații, este cel din listing 9.25. Prima procedură – `P_INIT_V_TRANSE_SV` – încarcă tranșele pentru determinarea procentului de spor de vechime din tabela `TRANSE_SV` în vectorul public `v_transe_sv`. Artificiul este binevenit, deoarece aceste tranșe sunt constante în timp, iar folosirea vectorului aduce un plus de viteză, sesizabil la dimensiuni mari ale tabelelor `PERSONAL` și `PONTAJE`. Variabila `i`, care apare în secțiunea declarativă a procedurii, are regim de variabilă privată, fiind accesabilă numai în procedură. Și cursorul `C_PERS` folosit în procedura `P_INIT_VECTORI_PERSONAL` este unul privat, prin comparație cu `C_ORE` care e “bun al întregii sesiuni”.

Listing 9.25. Corpul pachetului `PACHET_SALARIZARE`

```
CREATE OR REPLACE PACKAGE BODY pachet_salarizare AS

-----
-- procedura de inițializare a tabloului asociativ V_TRANSE_SV
PROCEDURE p_init_v_transe_sv IS
    i PLS_INTEGER;
BEGIN
    IF v_transe_sv.COUNT = 0 THEN
        FOR rec_transe IN (SELECT * FROM transe_sv ORDER BY ani_limita_inf) LOOP
            i := v_transe_sv.COUNT + 1;
            v_transe_sv(i).ani_limita_inf := rec_transe.ani_limita_inf;
            v_transe_sv(i).ani_limita_sup := rec_transe.ani_limita_sup;
            v_transe_sv(i).procent_sv := rec_transe.procent_sv;
        END LOOP;
    END IF;
END p_init_v_transe_sv;

-----

PROCEDURE p_init_vectori_personal IS
    CURSOR c_pers IS
        SELECT marca, salorar, salorarco, datasv
        FROM personal ORDER BY marca;
BEGIN
```

```

FOR rec_pers IN c_pers LOOP
    v_personal (rec_pers.marca).salorar := rec_pers.salorar ;
    v_personal (rec_pers.marca).salorarco := rec_pers.salorarco ;
    v_personal (rec_pers.marca).datasv := rec_pers.datasv ;
END LOOP ;
END p_init_vectori_personal ;

-----

FUNCTION f_este_in_pontaje (marca_personal.marca%TYPE,
    data_pontaje.data%TYPE ) RETURN BOOLEAN
AS
    v_este BOOLEAN ;
    v_unu INTEGER ;
BEGIN
    -- folosim un bloc inclus
    BEGIN
        SELECT 1 INTO v_unu FROM pontaje WHERE marca=marca_ AND data=data_ ;
        v_este := TRUE ;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN -- nu există înregistrarea în PONTAJE
            v_este := FALSE ;
    END ;
    RETURN v_este ;
END f_este_in_pontaje ;

-----

FUNCTION f_este_in_sporuri (marca_personal.marca%TYPE,
    an_v_an%TYPE, luna_v_luna%TYPE ) RETURN BOOLEAN
AS
    v_unu NUMBER(1) ;
BEGIN
    SELECT 1 INTO v_unu FROM sporuri
        WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
    RETURN TRUE ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE ;
END f_este_in_sporuri ;

-----

FUNCTION f_este_in_retineri (marca_personal.marca%TYPE,
    an_v_an%TYPE, luna_v_luna%TYPE ) RETURN BOOLEAN
AS
    v_unu NUMBER(1) ;
BEGIN
    SELECT 1 INTO v_unu FROM retineri
        WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
    RETURN TRUE ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE ;
END f_este_in_retineri ;

-----

FUNCTION f_este_in_salarii (marca_personal.marca%TYPE,
    an_v_an%TYPE, luna_v_luna%TYPE ) RETURN BOOLEAN
AS
    v_unu NUMBER(1) ;
BEGIN
    SELECT 1 INTO v_unu FROM salarii

```

```

WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
RETURN TRUE ;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RETURN FALSE ;
END f_este_in_salarii ;

-----

FUNCTION f_anii_vechime (datasv_ personal.datasv%TYPE,
an_ IN salarii.an%TYPE, luna_ salarii.luna%TYPE
) RETURN transe_sv.ani_limita_inf%TYPE
AS
prima_zi DATE := TO_DATE('01/'||luna_|| '/'||an_, 'DD/MM/YYYY') ;
BEGIN
RETURN TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0) ;
END f_anii_vechime ;

-----

FUNCTION f_procent_spor_vechime (ani_transe_sv.ani_limita_inf%TYPE)
RETURN transe_sv.procent_sv%TYPE
AS
v_procent transe_sv.procent_sv%TYPE := 0 ;
BEGIN
-- înainte de consultarea vectorului, se verifică dacă e inițializat
IF v_transe_sv.COUNT = 0 THEN
p_init_v_transe_sv ;
END IF ;
-- determinarea procentului
FOR i IN 1..v_transe_sv.COUNT LOOP
IF ani_ >= v_transe_sv(i).ani_limita_inf AND ani_ < v_transe_sv(i).ani_limita_sup THEN
v_procent := v_transe_sv(i).procent_sv ;
EXIT ;
END IF ;
END LOOP ;
RETURN v_procent ;
END f_procent_spor_vechime ;

END pachet_salarizare ;

```

Ar mai fi de spus că am renunțat la “polimorfismul” funcției F_EXIS-TA_SP_RE_SA, pe care am înlocuit-o cu trei funcții “cuminți”: F_ESTE_IN_SPO-RURI, F_ESTE_IN_RETINERI și F_ESTE_IN_SALARII. În plus, funcția F_PRO-CENT_SPOR_VECHIME determină procentul sporului de vechime în funcție de anii de vechime “scanând” nu tabela TRANSE_SV, ci vectorul v_transe_sv. Scanarea este precedată de verificarea inițializării vectorului, prin testul IF v_transe_sv.COUNT = 0.

9.3.3. Apelul obiectelor din pachet

Mărețele realizări ale pachetului sunt puse în valoare de procedura P_ACT_SP_SA2 care se dorește a fi o versiune evoluată a procedurii cu același nume, dar fără 2-ul din coadă (listing 9.22) – vezi listing 9.26. La apelul dintr-un bloc situat în afara pachetului, numele obiectului din pachet trebuie obligatoriu

prefixat de numele pachetului, ca, de exemplu `pachet_salarizare.c_ore` sau `pachet_salarizare.v_personal` etc.

Listing 9.26. Procedură ce apelează obiecte din pachet

```

/* procedura de actualizare a tabelelor SPORURI și SALARII pe baza datelor din
   PONTAJE - folosind pachetul PACHET_SALARIZARE */
CREATE OR REPLACE PROCEDURE p_act_sp_sa2
(an_salarii.an%TYPE, luna_salarii.luna%TYPE)
AS
-- C_ORE este deja declarat în pachet și, astfel, a devenit public
v_spvech sporuri.spvech%TYPE ;
v_venitbaza salarii.venitbaza%TYPE ;
v_spnoapte sporuri.spnoapte%TYPE ;
v_sporuri salarii.sporuri%TYPE ;
BEGIN
-- se verifică dacă vectorul V_PERSONAL este inițializat
IF pachet_salarizare.v_personal.COUNT = 0 THEN
    pachet_salarizare.p_init_vectori_personal ;
END IF ;

FOR rec_ore IN pachet_salarizare.c_ore (an_, luna_) LOOP
    -- se verifică dacă numărul orelor lucrate este exagerat
    IF rec_ore.ore_l > 176 THEN
        RAISE pachet_salarizare.prea_multe_ore ;
    END IF ;

    v_venitbaza := ROUND( rec_ore.ore_l *
        NVL(pachet_salarizare.v_personal(rec_ore.marca).salorar,0) +
        rec_ore.ore_co *
        NVL(pachet_salarizare.v_personal(rec_ore.marca).salorarco,0),-2) ;

    v_spvech := ROUND(v_venitbaza * pachet_salarizare.f_procent_spor_vechime(
        pachet_salarizare.f_an_i_vechime (pachet_salarizare.v_personal
        (rec_ore.marca ).datasv , an_, luna_)) / 100, -3) ;

    v_spnoapte := ROUND(rec_ore.ore_n * pachet_salarizare.v_personal(
        rec_ore.marca).salorar * .15, -3) ;

    IF pachet_salarizare.f_este_in_sporuri (rec_ore.marca, an_, luna_) THEN
        -- se actualizeaza tabela SPORURI pentru angajatul curent
        UPDATE sporuri
        SET spvech = v_spvech, ore_noapte = rec_ore.ore_n, spnoapte = v_spnoapte
        WHERE marca=rec_ore.marca AND an=an_ AND luna=luna_ ;
    ELSE
        INSERT INTO sporuri VALUES (rec_ore.marca, an_, luna_,
            v_spvech, rec_ore.ore_n, v_spnoapte, 0) ;
    END IF ;

    -- se procedează analog pentru tabela SALARII
    IF pachet_salarizare.f_este_in_salarii (rec_ore.marca, an_, luna_) THEN
        UPDATE salarii
        SET ore_lucrate = rec_ore.ore_l, oreco = rec_ore.ore_co,
            venitbaza = v_venitbaza, sporuri =
            (SELECT spvech + spnoapte + altesp
             FROM sporuri WHERE an=an_ AND luna=luna_ AND
             marca = rec_ore.marca)
        WHERE marca=rec_ore.marca AND an=an_ AND luna=luna_ ;
    ELSE

```

```

INSERT INTO salarii VALUES (rec_ore.marca, an_, luna_, rec_ore.ore_l,
                             rec_ore.ore_co, v_venitbaza, (SELECT spvech + spnoapte + altesp
                             FROM sporuri WHERE an=an_ AND luna=luna_ AND
                             marca = rec_ore.marca), 0, 0);

END IF ;
END LOOP ;
COMMIT ;
EXCEPTION
WHEN pachet_salarizare.prea_multe_ore THEN
    RAISE_APPLICATION_ERROR (-20005, 'E ceva in neregula cu pontajele');
END p_act_sp_sa2 ;

```

Execuția procedurii începe cu verificarea inițializării tabloului care conține salariile orare și datele de calcul ale sporurilor de vechime pentru toți angajații – v_personal. Verificarea presupune numărarea componentelor tabloului asociativ (IF pachet_salarizare.v_personal.COUNT = 0). Dacă numărul este zero, se apelează procedura de inițializare - pachet_salarizare.p_init_vectori_personal. În continuare se deschide și parcurge, linie cu linie, cursorul public C_ORE. Cursorul este unul parametrizat, așa că, la deschidere, i se pasează valorile anului și lunii pentru care se centralizează orele lucrate. Variabila compozită în care se stochează înregistrarea curentă din cursor este una locală – rec_ore. Pentru fiecare angajat se verifică dacă numărul orelor lunare lucrate depășește valoarea 170, ceea ce ar fi o samavolnicie și duce la declanșarea excepției prea_multe_ore. Venitul de bază, sporul de vechime și sporul de noapte folosesc valorile din tabloul v_personal și apelează la funcțiile pachetului. Secțiunea “excepțională”, adică cea dedicată excepțiilor, preia eroarea prea_multe_ore și nu face mare lucru cu ea, deoarece declanșează eroarea -2005 care afișează un mesaj muștrător.

9.3.4. Supraîncărcarea procedurilor/ funcțiilor din pachete

În cadrul unui pachet este permisă supraîncărcarea procedurilor și funcțiilor, o facilitate mult gustată în programarea orientată pe obiecte care permite aplicarea unei aceleași operații asupra unor obiecte de tipuri diferite. Numele procedurilor/funcțiilor supraîncărcate fiind comun, diferențierea se realizează prin parametri. Să examinăm conținutul listingului 9.27.

Listing 9.27. Specificațiile “supraîncărcate” ale pachetului

```

CREATE OR REPLACE PACKAGE pachet_exista AS

-- prima formă verifică existența (mărcii) în PERSONAL
FUNCTION f_exista (
    marca_personal.marca%TYPE) RETURN BOOLEAN ;

-- a doua formă verifică existența (mărcii/zilei) în PONTAJE
FUNCTION f_exista (
    marca_personal.marca%TYPE,
    data_pontaje.data%TYPE) RETURN BOOLEAN ;

-- a treia formă verifică existența combinației (marcă/an/luna) într-una

```

```

-- din tabelele SPORURI, RETINERI, SALARII
FUNCTION f_exista (
    marca_personal.marca%TYPE, an_salarii.an%TYPE, luna_salarii.luna%TYPE,
    tabela_ VARCHAR2) RETURN BOOLEAN ;

END pachet_exista ;

```

Specificațiile pachetului PACHET_EXISTA conțin trei versiuni ale funcției F_EXISTA care diferă prin parametri de intrare. Prima versiune primește o marca și întoarce TRUE dacă marca respectivă există în tabela PERSONAL și FALSE în caz contrar. A doua versiune primește un parametru în plus, data_, și testează existența combinației (marca_, data_) în tabela PONTAJE. În fine, a treia variantă este și cea mai complexă, deoarece căutarea se face, la alegere, într-una din tabelele SPORURI, RETINERI sau SALARII. Corpul celor trei funcții este cel din listing 9.28.

Listing 9.28. Corpul pachetului ce conține funcții supraîncărcate

```

CREATE OR REPLACE PACKAGE BODY pachet_exista AS

    -- prima formă
    FUNCTION f_exista (
        marca_personal.marca%TYPE) RETURN BOOLEAN
    IS
        v_unu NUMBER(1);
    BEGIN
        SELECT 1 INTO v_unu FROM personal WHERE marca = marca_ ;
        RETURN TRUE ;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE ;
    END f_exista ;

    -- a doua formă
    FUNCTION f_exista (
        marca_personal.marca%TYPE,
        data_pontaje.data%TYPE) RETURN BOOLEAN
    IS
        v_unu NUMBER(1);
    BEGIN
        SELECT 1 INTO v_unu FROM pontaje WHERE marca = marca_ AND data=data_ ;
        RETURN TRUE ;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN FALSE ;
    END f_exista ;

    -- a treia formă
    FUNCTION f_exista (
        marca_personal.marca%TYPE, an_salarii.an%TYPE, luna_salarii.luna%TYPE,
        tabela_ VARCHAR2) RETURN BOOLEAN
    IS
        v_unu NUMBER(1);
    BEGIN
        CASE
            WHEN UPPER(tabela_) = 'SPORURI' THEN
                SELECT 1 INTO v_unu FROM sporuri

```



```

        WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
    WHEN UPPER(tabela_) = 'RETINERI' THEN
        SELECT 1 INTO v_unu FROM retineri
        WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
    WHEN UPPER(tabela_) = 'SALARII' THEN
        SELECT 1 INTO v_unu FROM salarii
        WHERE marca=marca_ AND an=an_ AND luna=luna_ ;
    END CASE ;
    RETURN TRUE ;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN FALSE ;
    END f_exista ;

    END pachet_exista ;

```

Ilustrarea modului în care se apelează oricare dintre cele trei variante ale funcției constituie subiectul blocului anonim din listing 9.29. Pentru un plus de claritate, blocul afișează rezultatele fiecărei execuții a funcției.

Listing 9.29. Apelul funcțiilor supraîncărcate

```

DECLARE
    v_marca personal.marca%TYPE ;
    v_data pontaje.data%TYPE ;
    v_an salarii.an%TYPE ;
    v_luna salarii.luna%TYPE ;
BEGIN
    v_marca := 101 ;
    -- cautare în PERSONAL
    IF pachet_exista.f_exista (v_marca) THEN
        DBMS_OUTPUT.PUT_LINE('In PERSONAL exista angajat cu marca ' || v_marca) ;
    ELSE
        DBMS_OUTPUT.PUT_LINE('In PERSONAL NU exista angajat cu marca ' || v_marca) ;
    END IF ;

    -- căutare în PONTAJE
    v_data := TO_DATE('07/01/2003','DD/MM/YYYY') ;
    IF pachet_exista.f_exista (v_marca, v_data) THEN
        DBMS_OUTPUT.PUT_LINE('In PONTAJE exista inregistrare pentru marca ' ||
            v_marca || ' si ziua ' || v_data) ;
    ELSE
        DBMS_OUTPUT.PUT_LINE('In PONTAJE NU exista inregistrare pentru marca ' ||
            v_marca || ' si ziua ' || v_data) ;
    END IF ;

    -- căutare în SPORURI, RETINERI, SALARII
    v_an := 2002 ;
    v_luna := 5 ;
    IF pachet_exista.f_exista (v_marca, v_an, v_luna, 'SALARII') THEN
        DBMS_OUTPUT.PUT_LINE('In SALARII exista inregistrare pentru marca ' ||
            v_marca || ', anul ' || v_an || ' si luna ' || v_luna) ;
    ELSE
        DBMS_OUTPUT.PUT_LINE('In SALARII NU exista inregistrare pentru marca ' ||
            v_marca || ', anul ' || v_an || ' si luna ' || v_luna) ;
    END IF ;
END ;

```

Lansarea în execuție a blocului anonim în SQL*Plus se “soldează” cu un rezultat precum cel din figura 9.16.

```
SQL> @F:\ORACLE_CARTE\CAP09_PL_SQL2\LISTING09_29.SQL
In PERSONAL exista angajat cu marca 101
In PONTAJE exista inregistrare pentru marca 101 si ziua 07-JAN-03
In SALARII NU exista inregistrare pentru marca 101, anul 2002 si luna 5

PL/SQL procedure successfully completed.

SQL> |
```

Figura 9.16. Lansarea și rezultatele blocului de apel al funcțiilor supraîncărcate

PL/SQL nu permite supraîncărcarea a două proceduri/funcții dacă parametrii acestora diferă numai prin nume sau mod (IN, OUT, IN OUT), altfel spus, macăr pentru unul dintre parametri tipul trebuie să fie diferit. Mai mult, tipul nu trebuie să fie din aceeași familie (de exemplu, tipul CHAR este din aceeași familie cu VARCHAR2). Restricția este valabilă și în privința rezultatului – două funcții nu pot fi supraîncărcate numai pentru că tipul returnat este diferit. Multe din aceste restricții se manifestă nu în momentul creării (compilerul PL/SQL fiind destul de tolerant), ci la execuție (apel).

9.3.5. Inițializarea pachetelor

La primul apel al unui obiect dintr-un pachet, pachetul este instanțiat, fiind încărcat de pe disc în memorie, rezervându-se spațiu în memorie pentru variabile și, dacă apelul se referă la o procedură/funcție, inițializarea continuă execuția codului compilat al procedurii/funcției. Fiecare sesiune are o copie proprie a variabilelor din pachet, așa încât sesiuni diferite ce folosesc acelaș pachet au rezervate zone de memorie diferite.

PL/SQL oferă posibilitatea existenței unui cod de inițializare, executat automat la primul apel al pachetului. Acest cod este un bloc descris în corpul pachetului, după detalierea tuturor procedurilor și funcțiilor, după modelul:

```
CREATE OR REPLACE PACKAGE BODY un_pachet
AS
...
.... funcții, proceduri...
...

BEGIN
    cod_de_inițializare

END un_pachet ;
```

Având în vedere că și funcțiile de căutare au fost transferate pachetului PACHET_CAUTARE, “rescriem” pachetul PACHET_SALARIZARE. Listing 9.30 conține noile specificații.

Listing 9.30. Noile specificații pentru PACHET_SALARIZARE

```

CREATE OR REPLACE PACKAGE pachet_salarizare AS

v_an salarii.an%TYPE := TO_NUMBER(TO_CHAR(SYSDATE, 'YYYY'));
v_luna salarii.luna%TYPE := TO_NUMBER(TO_CHAR(SYSDATE, 'MM'));

-- variabila pentru păstrarea liniilor din TRANSE_SV
TYPE t_transe_sv IS TABLE OF transe_sv%ROWTYPE INDEX BY PLS_INTEGER;
v_transe_sv t_transe_sv;

-- procedura de inițializare a vectorului asociativ V_TRANSE_SV
PROCEDURE p_init_v_transe_sv;

/* se declară un vector asociativ pentru datele din PERSONAL */
TYPE r_personal IS RECORD (
    salorar personal.salorar%TYPE, salorarCO personal.salorarCO%TYPE,
    datasv personal.datasv%TYPE);
TYPE t_personal IS TABLE OF r_personal INDEX BY PLS_INTEGER;
v_personal t_personal;

PROCEDURE p_init_vectori_personal;

/* funcțiile de căutare sunt acum în pachetul PACHET_EXISTA */

FUNCTION f_an_i_vechime (datasv_personal.datasv%TYPE,
    an_IN salarii.an%TYPE, luna_salarii.luna%TYPE
) RETURN transe_sv.ani_limita_inf%TYPE;

FUNCTION f_procent_spor_vechime (ani_transe_sv.ani_limita_inf%TYPE)
    RETURN transe_sv.procent_sv%TYPE;

CURSOR c_ore (an_v_an%TYPE, luna_v_luna%TYPE) IS
    SELECT marca, SUM(orelucrate) AS ore_l, SUM(oreco) AS ore_co,
    SUM(orenoapte) AS ore_n FROM pontaje
    WHERE TO_NUMBER(TO_CHAR(data, 'YYYY')) = an AND
    TO_NUMBER(TO_CHAR(data, 'MM')) = luna GROUP BY marca;

prea_multe_ore EXCEPTION;
END pachet_salarizare;

```

Față de versiunea anterioară (listing 9.24), au dispărut funcțiile F_ESTE_IN_PONTAJE, F_ESTE_IN_SPORURI, F_ESTE_IN_RETINERI și F_ESTE_IN_SALARII. În rest... nimic nou sub soare ! Corpul pachetului – listing 9.31 – aduce totuși câteva elemente de noutate. Mai întâi, în procedura de inițializare a tabloului asociativ v_transe_sv nu se mai testează dacă numărul componentelor vectorului este zero, ci se șterg toate elementele și apoi se populează tabloul. Analog se procedează cu v_personal în procedura P_INIT_VECTORI_PERSONAL.

Listing 9.31. Corpul pachetului, în care este inclus codul de inițializare

```

-- corpul pachetului, plus codul de inițializare
CREATE OR REPLACE PACKAGE BODY pachet_salarizare AS
-----

-- procedura de inițializare a tabloului asociativ V_TRANSE_SV
PROCEDURE p_init_v_transe_sv IS
    i PLS_INTEGER;

```

```

BEGIN
    v_transe_sv.DELETE ;
    FOR rec_transe IN (SELECT * FROM transe_sv ORDER BY ani_limita_inf) LOOP
        i := v_transe_sv.COUNT + 1 ;
        v_transe_sv(i).ani_limita_inf := rec_transe.ani_limita_inf ;
        v_transe_sv(i).ani_limita_sup := rec_transe.ani_limita_sup ;
        v_transe_sv(i).procent_sv := rec_transe.procent_sv ;
    END LOOP ;
END p_init_v_transe_sv ;

-----

PROCEDURE p_init_vectori_personal
IS
    CURSOR c_pers IS SELECT marca, salorar, salorarco, datasv
    FROM personal ORDER BY marca ;
BEGIN
    v_personal.DELETE ;
    FOR rec_pers IN c_pers LOOP
        v_personal(rec_pers.marca).salorar := rec_pers.salarar ;
        v_personal(rec_pers.marca).salorarco := rec_pers.salorarco ;
        v_personal(rec_pers.marca).datasv := rec_pers.datasv ;
    END LOOP ;
END p_init_vectori_personal ;

-----

FUNCTION f_anii_vechime (datasv_personal.datasv%TYPE,
    an_ IN salarii.an%TYPE, luna_salarii.luna%TYPE
    ) RETURN transe_sv.ani_limita_inf%TYPE
AS
    prima_zi DATE := TO_DATE('01/'||luna_||'/'||an_, 'DD/MM/YYYY') ;
BEGIN
    RETURN TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0) ;
END f_anii_vechime ;

-----

FUNCTION f_procent_spor_vechime (ani_transe_sv.ani_limita_inf%TYPE)
    RETURN transe_sv.procent_sv%TYPE
AS
    v_procent transe_sv.procent_sv%TYPE := 0 ;
BEGIN
    -- ...v_transe_sv e cu siguranta initializat ..

    -- determinarea procentului
    FOR i IN 1..v_transe_sv.COUNT LOOP
        IF ani_ >= v_transe_sv(i).ani_limita_inf AND
            ani_ < v_transe_sv(i).ani_limita_sup THEN
            v_procent := v_transe_sv(i).procent_sv ;
            EXIT ;
        END IF ;
    END LOOP ;
    RETURN v_procent ;
END f_procent_spor_vechime ;

-- aici începe codul de inițializare !!!
BEGIN
    p_init_v_transe_sv ;
    p_init_vectori_personal ;

END pachet_salarizare ;

```

Mult lăudatul cod de inițializare se găsește, discret, în finalul corpului pachetului. Practic, la prima invocare a pachetului se populează cei doi vectori, astfel încât subprogramele ce fac apel la aceștia să nu mai testeze în prealabil dacă există componente în tablourile respective. Pentru un plus de claritate, procedura P_ACT_SP_SA2 (listing 9.26) se modifică – vezi listing 9.32.

Listing 9.32. Versiune modificată a procedurii ce apelează obiecte din pachet

```

/* procedura de actualizare SPORURI/SALARII - altă versiune */
CREATE OR REPLACE PROCEDURE p_act_sp_sa2 (an_salarii.an%TYPE,
      luna_salarii.luna%TYPE)
AS
    ... secțiunea declarativă e neschimbată

BEGIN
    -- nu mai trebuie să se verifice dacă V_PERSONAL este inițializat

    FOR rec_ore IN pachet_salarizare.c_ore (an_, luna_) LOOP
        -- se verifica daca numarul orelor lucrate este exagerat
        IF rec_ore.ore_l > 190 THEN
            RAISE pachet_salarizare.prea_multe_ore ;
        END IF ;

        ... expresiile pentru calculul v_venit_baza, v_spvech și v_spnoapte rămân aceleași

        IF pachet_exista.f_exista (rec_ore.marca, an_, luna_, 'SPORURI') THEN
            ... nici aici nu apar modificări
        END IF ;

        -- se procedeaza analog pentru tabela SALARII
        IF pachet_exista.f_exista (rec_ore.marca, an_, luna_, 'SALARII') THEN
            ... nici aici nu apar modificări
        END IF ;
    END LOOP ;
    COMMIT ;
EXCEPTION
    ... identic listingului 9.26
END p_act_sp_sa2 ;
/

```

9.3.6. Informații despre pachete în dicționarul bazei

Primele tabele virtuale vizate în dicționar pentru a afla câte ceva despre pachete este USER_OBJECTS și USER_SOURCE. Dacă ne interesează ce pachete există în schema curentă și starea lor (dacă prezintă sau nu erori la compilare), se folosește fraza SELECT:

```

SELECT object_name, status
FROM user_objects
WHERE object_type = 'PACKAGE'

```

Interogarea afișează pachetele ce prezintă specificații. După cum am văzut, pot exista pachete fără corp, așa încât afișarea acestora prespune interogarea:

```

SELECT object_name, status

```

```

FROM user_objects
WHERE object_type = 'PACKAGE' AND object_name NOT IN
  (SELECT object_name
   FROM user_objects
   WHERE object_type = 'PACKAGE BODY')

```

Conținutul specificațiilor se realizează de o manieră similară procedurilor și funcțiilor:

```

SELECT text
FROM user_source
WHERE name = 'PACHET_SALARIZARE' AND type='PACKAGE'
ORDER BY line

```

iar pentru obținerea liniilor ce alcătuiesc corpul pachetului se modifică clauza WHERE:

```

SELECT text
FROM user_source
WHERE name = 'PACHET_SALARIZARE' AND type='PACKAGE BODY'
ORDER BY line

```

Pe lângă acestea, uneori este utilă și USER_PROCEDURES. La prima vedere coloanele acesteia pot fi înșelătoare, deoarece dintre OBJECT_NAME și PROCEDURE_NAME, prima conține numele procedurilor/funcțiilor “independente” din schemă, în timp ce a doua indică subprogramele create în cadrul unui pachet. Așa că pentru a afișa, în același raport, și pe cele incluse în pachete și pe cele independente, se poate recurge la interogarea:

```

SELECT
  CASE
    WHEN procedure_name IS NOT NULL THEN object_name
    ELSE '-proc./functie independenta-'
  END AS pachet,
  NVL(procedure_name, object_name) AS procedura_functie
FROM user_procedures

```

Rezultatul este cel din figura 9.17.

PACHET	PROCEDURA_FUNCȚIE
-proc./funcție independentă-	EROARE_CONTROLATA
-proc./funcție independentă-	F_AFLA_SALORAR
-proc./funcție independentă-	F_ANI_UECHIME
-proc./funcție independentă-	F_CAUTA_MARCA
-proc./funcție independentă-	F_EC2
-proc./funcție independentă-	F_ESTE_IN_PONTAJE
-proc./funcție independentă-	F_EXISTA
-proc./funcție independentă-	F_EXISTA_SP_RE_SA
-proc./funcție independentă-	F_PERSONAL
-proc./funcție independentă-	F_PROCENT_SPOR_UECHIME
-proc./funcție independentă-	F_SALORAR
-proc./funcție independentă-	ORDONARE_5
-proc./funcție independentă-	ORDONARE_5U2
PACHET_EXISTA	F_EXISTA
PACHET_EXISTA	F_EXISTA
PACHET_EXISTA	F_EXISTA
PACHET_SALARIZARE	F_ANI_UECHIME
PACHET_SALARIZARE	F_PROCENT_SPOR_UECHIME
PACHET_SALARIZARE	P_INIT_VECTORI_PERSONAL
PACHET_SALARIZARE	P_INIT_V_TRANSE_SU
-proc./funcție independentă-	P_ACT_SP_SA
-proc./funcție independentă-	P_ACT_SP_SA2
-proc./funcție independentă-	P_EC2
-proc./funcție independentă-	P_NO_COPY
-proc./funcție independentă-	P_POPULARE_PONTAJE
-proc./funcție independentă-	P_POPULARE_PONTAJE2
-proc./funcție independentă-	P_POPULARE_PONTAJE_AN
-proc./funcție independentă-	P_POPULARE_PONTAJE_LUNA

28 rows selected.

SQL>

Figura 9.17. Afișarea tuturor procedurilor și funcțiilor

9.4. Dependente între blocurile numite

Comenzile `DROP`, ca, de altfel, și comenzile `CREATE` OR `REPLACE` pentru funcții sau proceduri și comenzile `ALTER TABLE` sau `ALTER VIEW` pentru tabele și view-uri, pot avea însă efecte directe și indirecte și asupra altor obiecte din baza de date. Spre exemplu, dacă într-o procedură este declarat un cursor a cărui frază `SELECT` vizează o anumită tabelă, atunci eliminarea sau doar modificarea respectivei tabele poate invalida procedura care o invocă.

Posibilitatea apariției acestor efecte ca urmare a dependențelor dintre obiecte este logică, dar serverul Oracle nu așteaptă ca ele să producă erori „fatale” în timpul execuției sau invocării obiectelor dependente. Îndată ce a avut loc un astfel de eveniment care poate avea repercursiuni asupra relațiilor de dependență, serverul „marchează” acele obiecte în dicționarul bazei ca invalide, astfel încât să necesite o recompilare. Prin urmare, inarvertențele datorate „alterării” dependențelor dintre obiecte vor apare în momentul recompilării și nu al execuției.

9.4.1. Ștergerea blocurilor numite

Eliminarea unei proceduri de sine stătătoare, sau a unei funcții de acest fel, se face printr-o comandă `DROP PROCEDURE` sau `DROP FUNCTION`, ce primește ca argument numele obiectului vizat. Dacă procedura sau funcția respectivă este definită în interiorul unui pachet, atunci renunțarea la ea nu se poate face decât prin redefinirea pachetului care să elimine, din specificațiile sale, declarațiile acesteia. Bineînțeles ștergerea unui pachet implică ștergerea membrilor acestuia.

În cazul pachetelor, eliminarea lor se realizează prin comanda `DROP PACKAGE`, ceea ce înseamnă renunțarea atât la antetul pachetului cât și la „corpul” său. Dacă însă se dorește numai ștergerea implementării din „body” a unui pachet, atunci comanda va lua forma `DROP PACKAGE BODY`.

În general, posibilitatea creării modulelor procedurale prin comenzi `CREATE OR REPLACE` reduce numărul situațiilor în care este necesară folosirea explicită a unor comenzi `DROP` doar la cazurile în care, într-adevăr, ceea ce se intenționează echivalează cu renunțarea definitivă la respectivul obiect.

9.4.2. Dependențe dintre proceduri/funcții/pachete

Dependențele directe dintre obiecte le putem afla prin intermediul view-ului `USER_DEPENDENCIES` care are structura următoare (figura 9.18):

SQL> desc user_dependencies		
Name	Null?	Type
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(17)
REFERENCED_OWNER		VARCHAR2(30)
REFERENCED_NAME		VARCHAR2(64)
REFERENCED_TYPE		VARCHAR2(17)
REFERENCED_LINK_NAME		VARCHAR2(128)
SCHEMAID		NUMBER
DEPENDENCY_TYPE		VARCHAR2(4)

Figura 9.18. Structura view-ului `USER_DEPENDENCIES`

Să ne orientăm către un exemplu concret. Luăm în considerare procedura `P_ACT_SP_SA2` și verificăm mai întâi starea acesteia astfel:

SQL> select object_name, status from user_objects where object_name = 'P_ACT_SP_SA2';	
OBJECT_NAME	STATUS
P_ACT_SP_SA2	VALID

Figura 9.19. Starea inițială a procedurii `P_ACT_SP_SA2`

Dependențele acestei proceduri pot fi obținute printr-o interogare de genul celei din figura 9.20:


```
SQL> select name, referenced_name, referenced_type
       2 from user_dependencies where name = 'P_ACT_SP_SA2';
```

NAME	REFERENCED_NAME	REFERENCED_TYPE
P_ACT_SP_SA2	STANDARD	PACKAGE
P_ACT_SP_SA2	DBMS_STANDARD	PACKAGE
P_ACT_SP_SA2	PLITBLM	PACKAGE
P_ACT_SP_SA2	PLITBLM	SYNONYM
P_ACT_SP_SA2	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
P_ACT_SP_SA2	SPORURI	TABLE
P_ACT_SP_SA2	SALARII	TABLE
P_ACT_SP_SA2	TRANSE_SU	TABLE
P_ACT_SP_SA2	PACHET_SALARIZARE	PACKAGE
P_ACT_SP_SA2	PLITBLM	NON-EXISTENT
P_ACT_SP_SA2	PACHET_EXISTA	PACKAGE

Figura 9.20. Dependențele procedurii P_ACT_SP_SA2

Procedura de mai sus invocă, la un moment dat, în definiția ei pachetul PACHET_SALARIZARE, lucru perfect adevărat dacă de gândim că pentru a actualiza tabela SPORURI este parcurs cursorul C_ORE a cărui definiție o regăsim în PACHET_SALARIZARE (vezi listing 9.32). Să refacem acest pachet reexecutând instrucțiunile din listing 9.30, după care să verificăm din nou starea procedurii.

```
SQL> select object_name, status from user_objects where object_name = 'P_ACT_SP_SA2';
```

OBJECT_NAME	STATUS
P_ACT_SP_SA2	INVALID

Figura 9.21. Invalidarea procedurii P_ACT_SP_SA2

Se observă că starea procedurii este modificată și, ca urmare, fie implicit la prima execuție, fie explicit prin comanda:

```
ALTER PROCEDURE p_act_sp_sa2 COMPILE;
```

aceasta va trebui recompilată pentru a o revalida.

În cazul dependenței prezentate mai sus, lucrurile sunt „oarecum” clare și ușor de înțeles. Să mai facem totuși un experiment: modifică structura tabeli PONTAJE și apoi verificăm starea procedurii P_ACT_SP_SA2:

```
SQL> ALTER TABLE pontaje MODIFY oreabsnem NUMBER(3);
```

Table altered.

```
SQL> col object_name FORMAT a30
```

```
SQL> select object_name, status from user_objects where object_name = 'P_ACT_SP_SA2';
```

OBJECT_NAME	STATUS
P_ACT_SP_SA2	INVALID

Figura 9.22. Invalidarea (din nou) a procedurii P_ACT_SP_SA2

Dacă privim dependențele enumerate în figura 9.21, nu regăsim în coloana `REFERENCED_NAME` numele tabelului `PONTAJE`. Prin urmare, ce s-ar fi putut întâmpla așa încât să producă invalidarea procedurii `P_ACT_SP_SA2`? Răspunsul l-am putea schematiza ca în figura 9.23. Invalidarea unui obiect poate avea loc chiar și datorită dependențelor indirecte, nerelevante explicit printr-o linie în tabela virtuală `USER_DEPENDENCIES`. În cazul concret al procedurii `P_ACT_SP_SA2` explicația este următoarea: modificarea definiției tabelului `PONTAJE` afectează definiția cursorului `C_ORE` pe care îl găsim în pachetul `PACHET_SALARIZARE`, fapt ce va produce invalidarea acestuia. Mergând apoi pe linia dependenței directe dintre pachet și procedura amintită, se va produce și invalidarea acesteia din urmă. Ca urmare, pentru a restabili legitimitatea, va trebui să executăm ambele instrucțiuni de recompilare:

```
ALTER PACKAGE pachet_salarizare;  
ALTER PROCEDURE p_act_sp_sa2;
```

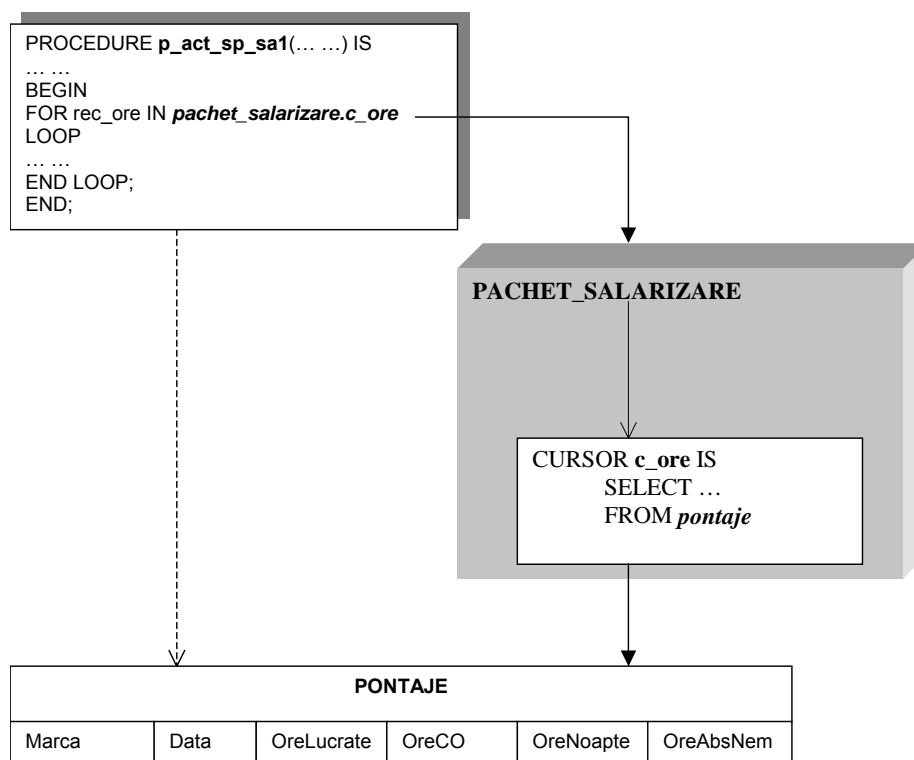


Figura 9.22. Dependența (indirectă) a procedurii `P_ACT_SP_SA2` cu tabela `PONTAJE`

Același efect s-ar putea produce și dacă în locul pachetului ar fi fost o funcție, o altă procedură sau chiar un view. Pentru a ierarhiza dependențele directe și indirecte putem încerca o frază `SELECT` de genul:

```
SELECT LPAD(' ', 2*(LEVEL-1)) || name dependencies_chart,
```

```

        referenced_name
FROM user_dependencies
WHERE referenced_name IN
      (SELECT object_name FROM user_objects)
START WITH name = %nume_procedură%
CONNECT BY name = PRIOR referenced_name AND
        name <> referenced_name

```

Dacă am înlocui %nume_procedură% cu P_ACT_SP_SA2, rezultatul ar fi cel din figura 9.23. După cum se vede, chiar și modificarea structurii tabeli RETINERI ar produce invalidarea P_ACT_SP_SA2. Acest lucru nu este valabil însă decât într-o singură situație: dacă dependența dintre PACHET_EXISTA și tabela RETINERI se realizează la nivelul antetului, nu (sau nu numai) la nivelul corpului procedural (*package body*).

```

SQL> set pagesize 100
SQL> SELECT LPAD(' ',2*(LEVEL-1)) || name dependencies_chart, referenced_name
 2 FROM user_dependencies
 3 WHERE referenced_name IN (select object_name from user_objects)
 4 START WITH name = 'P_ACT_SP_SA2'
 5 CONNECT BY name = PRIOR referenced_name AND name <> referenced_name
 6 /

```

DEPENDENCIES_CHART	REFERENCED_NAME
P_ACT_SP_SA2	PACHET_EXISTA
PACHET_EXISTA	SPORURI
PACHET_EXISTA	SALARII
PACHET_EXISTA	SALARII
PACHET_EXISTA	RETINERI
PACHET_EXISTA	PONTAJE
PACHET_EXISTA	PERSONAL
PACHET_EXISTA	PERSONAL
PACHET_EXISTA	PONTAJE
P_ACT_SP_SA2	PACHET_SALARIZARE
PACHET_SALARIZARE	TRANSE_SU
PACHET_SALARIZARE	TRANSE_SU
PACHET_SALARIZARE	SALARII
PACHET_SALARIZARE	SALARII
PACHET_SALARIZARE	PONTAJE
PACHET_SALARIZARE	PERSONAL
PACHET_SALARIZARE	PERSONAL
P_ACT_SP_SA2	TRANSE_SU
P_ACT_SP_SA2	SALARII
P_ACT_SP_SA2	SPORURI

20 rows selected.

Figura 9.23. Dependențele directe și indirecte ale procedurii P_ACT_SP_SA2

Astfel, dacă modificăm definiția PACHET_EXISTA din listing 9.29 și adăugăm definiția unei variabile, să zicem

```

TYPE rec_retineri IS RECORD (altele retineri.alteret%TYPE);

```

atunci am putea obține situația din figura 9.24:

```

SQL> select object_name, status from user_objects where object_name = 'P_ACT_SP_SA2';

OBJECT_NAME          STATUS
-----
P_ACT_SP_SA2         VALID

SQL> alter table RETINERI modify ALTERET NUMBER(17, 2);

Table altered.

SQL> select object_name, status from user_objects where object_name = 'P_ACT_SP_SA2';

OBJECT_NAME          STATUS
-----
P_ACT_SP_SA2         INVALID

SQL> select object_name, status from user_objects where object_name = 'PACHET_EXISTA';

OBJECT_NAME          STATUS
-----
PACHET_EXISTA        INVALID
PACHET_EXISTA        INVALID

```

Figura 9.24. Invalidarea procedurii P_ACT_SP_SA2 pe baza unei dependențe indirecte

Analizând rezultatul din figura 9.24 putem concluziona că nu toate dependențele față de pachete produc invalidarea obiectelor dependente, ci doar acele dependențe care sunt legate de declarațiile din antetele acelor pachete. Cu alte cuvinte, invalidarea corpului procedural al unui pachet nu produce invalidarea obiectelor care nu depind de fapt de antetul acestuia.

Pentru mai multă exactitate în această privință, serverul Oracle oferă posibilitatea obținerii informațiilor despre referințe într-o formă ierarhică mai bine organizată prin intermediul unui sistem de tabele simple și tabele virtuale create prin script-ul `utldtree.sql` care se găsește în directorul `rdbms/admin` al reședinței de instalare a serverului, de exemplu

```
@ c:\Oracle\Ora92\Rdbms\Admin\utldtree.sql
```

După lansarea acestui script, se populează mai întâi tabela care conține ierarhia de dependențe, pornind de la obiectul ale cărui obiecte dependente dorim să le aflăm. Apoi, prin invocarea procedurii `DEPTREE_FEEL` cu trei argumente: tipul obiectului, numele schemei în care se găsește obiectul și numele acestuia putem obține o nouă ierarhie de dependențe. De exemplu, pentru `PACHET_EXISTA` vom executa:

```
EXECUTE deptree_fill('PACKAGE', 'PERSONAL',
                    'PACHET_EXISTA')
```

După care, dacă vom lansa următoarea interogare:

```
SELECT nested_level, type, name from deptree order by seq#;
```

vom obține rezultatul din figura 9.25.

```
SQL> select nested_level, type, name from deptree order by seq#;
```

NESTED_LEVEL	TYPE	NAME
0	PACKAGE	PACHET_EXISTA
1	PACKAGE BODY	PACHET_EXISTA
1	PROCEDURE	P_ACT_SP_SA2

Figura 9.25. Obiectele dependente de PACHET_EXISTA

Dacă repetăm același experiment cu PACHET_EXISTA ca PACKAGE_BODY:

```
EXECUTE deptree_fill('PACKAGE BODY', 'PERSONAL',
'PACHET_EXISTA')
```

SELECT nested_level, type, name from deptree order by seq#;
vom obține rezultatul din figura 9.26.

```
SQL> SELECT nested_level, type, name from deptree order by seq#;
```

NESTED_LEVEL	TYPE	NAME
0	PACKAGE BODY	PACHET_EXISTA

Figura 9.26. Obiectele dependente de corpul pachetului PACHET_EXISTA

Prin urmare, P_ACT_SP_SA2 depinde de antetul pachetului PACHET_EXISTA și nu de corpul său.

9.5. Alte opțiuni PL/SQL

PL/SQL este un limbaj complet și, în același timp, complex. Pe lângă caracteristicile de bază, tratate până în prezent, mai găsim o serie de facilități legate de anumite situații speciale, între care: drepturile și contul sub care vor fi executate modulele procedurale (atunci când dreptul de invocare a acestora este transmis și altor utilizatori în afară de contul în care au fost create inițial), declarații de variabile gen colecții care să facă referire la „obiecte” de tip cursor (nu vectori asociativi sau tabele încapsulate), „încapsularea” unor „zone” tranzacționale izolate într-un nivel separat față de tranzacția curentă sau posibilitatea de a efectua în interiorul funcțiilor și operații care să producă modificări asupra unor obiecte din baza de date, nu numai să returneze pur și simplu valori.

9.5.1. Drepturi de invocare

Maniera în care am creat procedurile, funcțiile și pachetele până în momentul de față impune ca, pentru execuția normală a acestora, contul sub care au fost create să aibă privilegii corespunzătoare în privința referințelor exterioare, mai exact asupra elementelor din afara definițiilor procedurale, implicate în prelucrările interne ale acestor obiecte sau în declarațiile tipurilor variabilelor. De exemplu, în cazul procedurii P_ACT_SP_SA2, utilizatorul care deține această

funcție (proprietarul schemei PERSONAL) trebuie să aibă drepturi de SELECT, INSERT, UPDATE etc. asupra tabelor SALARII și SPORURI. Acest lucru este inherent din moment ce respectivele tabele se găsesc tot în schema deținută de proprietarul procedurii P_ACT_SP_SA2. Problema care se pune este următoarea: această procedură ar putea fi executată de un alt utilizator (deținător al altei scheme) ? Și dacă acest lucru este posibil, atunci în momentul în care respectivul utilizator execută procedura, ale cui drepturi sunt implicate ?

Serverul Oracle permite acordarea drepturilor de execuție (sau invocare) asupra obiectelor procedurale și altor utilizatori, acțiune efectuată de regulă de către deținătorul schemei ce conține respectivele obiecte. În momentul sunt invocării, acestea vor fi executate implicit „sub contul” în care au fost create, pe baza drepturilor utilizatorului proprietar, iar referințele relative (obiectele necalificate prin numele schemei din care fac parte) sunt rezolvate în schema acestuia. Această stare de fapt are cel puțin două implicații: în primul rând, respectivele proceduri nu pot fi reutilizate asupra obiectelor din alte scheme (tabele,view-uri etc.) și, ca urmare, partajarea codului în acest caz nu este posibilă, iar în al doilea rând se ridică și o problemă legată de securitate – deși nu vor avea drepturi directe asupra tabelor sau altor obiecte implicate în prelucrările procedurilor invocate, utilizatorii cărora li s-au acordat drepturi de execuție pot efectua operații asupra datelor care nu le aparțin, iar aceste modificări sunt consemnate sub contul proprietar și nu sub conturile acestora.

Să luăm un exemplu. Mai întâi, pe lângă contul inițial PERSONAL care deține procedura P_ACT_SP_SA2, mai creăm un cont, PERSONAL_2, astfel:

```
CREATE USER personal_2 IDENTIFIED BY personal_2
      DEFAULT TABLESPACE users
      TEMPORARY TABLESPACE temp;
GRANT connect, resource TO personal_2;
```

după care, dintr-o sesiune deschisă folosind contul PERSONAL, acordăm dreptul de execuție asupra procedurii P_ACT_SP_SA2

```
GRANT execute ON p_act_sp_sa2 TO personal_2;
```

Inițial, în tabela SPORURI presupunem că situația ar fi cea din figura 9.27:

```
SQL> SELECT * FROM sporuri;
```

MARCA	AN	LUNA	SPUECH	ORENOAPTE	SPNOAPTE	ALTESP
1001	2003	3	0	0	0	0
1002	2003	3	0	0	0	0
1003	2003	3	0	0	0	0
1004	2003	3	0	0	0	0

Figura 9.27. Situația inițială în tabela SPORURI

Deschidem apoi o sesiune separată folosind contul PERSONAL_2, și încercăm să accesăm tabela SPORURI din schema PERSONAL. Evident, neavând drepturi pe această tabela vom obține un mesaj prin care se transmite că

PERSONAL.SPORURI „nu există” în ceea ce îl privește pe utilizatorul PERSONAL_2 (vezi figura 9.27).

```
SQL> CONNECT personal_2/personal_2
Connected.
SQL> SELECT * FROM personal.sporuri;
SELECT * FROM personal.sporuri
*
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> UPDATE personal.sporuri SET spvech = 1500000;
UPDATE personal.sporuri SET spvech = 1500000
*
ERROR at line 1:
ORA-00942: table or view does not exist

SQL> |
```

Figura 9.28. PERSONAL_2 nu are drepturi asupra tabeli PERSONAL.SPORURI

Din sesiunea deschisă prin contul PERSONAL2 executăm procedura P_ACT_SP_SA2 din schema PERSONAL:

```
EXECUTE personal.p_act_sp_sa2(2003, 3)
```

Verificăm apoi, din sesiunea deschisă prin contul PERSONAL, datele din tabela SPORURI (vezi figura 9.27). Vom observa modificările efectuate de către utilizatorul PERSONAL2 care are drepturi de execuție pentru procedura P_ACT_SP_SA2 și nu a avut nevoie explicit de drepturi pentru UPDATE pe tabela SPORURI din schema PERSONAL.

```
SQL> select * from sporuri;
```

MARCA	AN	LUNA	SPUECH	ORENOAPTE	SPNOAPTE	ALTESP
1001	2003	3	25000	0	0	0
1002	2003	3	40000	4	30000	0
1003	2003	3	156000	8	78000	0
1004	2003	3	0	4	24000	0

Figura 9.29. Modificările operate în tabela SPORURI de către utilizatorul PERSONAL2

Există situații în care partajarea logicii „stocate” în baza de date între mai multe scheme este utilă. De exemplu, presupunem că aplicația noastră privind salariile are atât de mult succes încât se dorește a fi implementată de către o firmă de talie cel puțin „națională”. Respectiva firmă posedă o structură de organizare bazată pe mai multe filiale puternice răspândite în diverse locații în țară. Conducerea preferă folosirea unei singure baze de date centralizată în care fiecare filială să-și gestioneze datele despre salarii în propria „schemă”. În această ipoteză va trebui să „clonăm” structura modulară a aplicației pentru fiecare schemă în parte. Poate că acest lucru nu reprezintă în sine o idee nefericită, însă, legat de repetatele

operațiuni de instalare și actualizare a logicii ca urmare a schimbărilor majore ce pot apare la nivelul unui număr mai mic sau mai mare de module, costurile întreținerii ar crește proporțional cu numărul de „clone”. Soluția ar fi instalarea logicii „stocate” din baza de date într-o schemă „neutră” din care să fie invocată de conturile corespunzătoare fiecărei filiale în parte, iar modificările să se răsfrângă asupra tabelor din respectivele scheme.

Cu alte cuvinte, vom dori ca procedura P_ACT_SP_SA2 să fie executată folosind drepturile contului care o invocă, iar referințele externe să fie rezolvate în schema acestui cont și nu sub schema contului în care este creată. Pentru ca acest lucru să fie posibil este necesar ca această procedură să fie refăcută astfel încât comanda CREATE OR REPLACE PROCEDURE să includă și clauza **AUTHID CURRENT USER** ceea ce implică execuția acesteia sub contul (schema) din care a fost invocată. Implicit clauza AUTHID (aplicabilă bineînțeles și pentru CREATE OR REPLACE FUNCTION, sau PACKAGE sau TYPE) este specificată cu DEFINER ceea ce conduce la situația ce permite invocarea cu drepturile utilizatorului proprietar. Pentru a materializa o astfel de ipoteză de lucru să creăm un nou cont PERSONAL_MASTER astfel:

```
CREATE USER personal_master IDENTIFIED BY personal_master
  DEFAULT TABLESPACE users
  TEMPORARY TABLESPACE temp;
GRANT connect, resource TO personal_master;
```

Această nouă schemă constituie depozitul pentru logica stocată în baza de date care va fi partajată de toate schemele interesate. Prin urmare, sub acest cont vom reface procedura P_ACT_SP_SA2 modificând scriptul din listing 9.33 astfel:

Listing 9.33. Versiune modificată a procedurii P_ACT_SP_SA2 pentru a putea fi invocată cu drepturile utilizatorului curent

```
/* procedura de actualizare SPORURI/SALARII - versiune partajabilă */
CREATE OR REPLACE PROCEDURE p_act_sp_sa2 (an_salarii.an%TYPE,
  luna_salarii.luna%TYPE)
AUTHID CURRENT_USER AS
... celelalte secțiuni rămân neschimbate
```

După cum am văzut în paragraful anterior, procedura P_ACT_SP_SA2 depinde de pachetele PACHET_EXISTA și PACHET_SALARIZARE. Dacă în schema PERSONAL_MASTER vom crea cele două pachete așa cum sunt definite în listing 9.27 și listing 9.30, atunci am „viciat” din start rezultatul final. Aceasta deoarece într-o secvență de apeluri care începe cu o procedură ce beneficiază de „invoker rights” (execuție cu drepturile utilizatorului curent), dacă unul dintre apeluri vizează o procedură creată cu „definer rights” (execuție cu drepturile utilizatorului în schema în care a fost definită), atunci respectivul apel este rezolvat cu drepturile locale ale schemei în care se găsește respectiva procedură și, după încheierea derulării acestui pas, secvența inițială continuă folosind drepturile utilizatorului curent. Prin urmare cele două scripturi trebuie modificate după cum urmează:

Listing 9.34. Specificațiile “supraîncărcate” ale pachetului

```
CREATE OR REPLACE PACKAGE pachet_exista AUTHID CURRENT_USER AS
... celelalte declarații rămân neschimbate
```

Listing 9.35. Noile specificații pentru PACHET_SALARIZARE

```
CREATE OR REPLACE PACKAGE pachet_salarizare AUTHID CURRENT_USER AS
... celelalte declarații rămân neschimbate
```

De asemenea, pentru ca procedura P_ACT_SP_SA2 și aceste două pachete să fie compilate corespunzător, este necesar ca în schema PERSONAL_MASTER să fie create și tabelele de care acestea depind, chiar dacă ele nu vor fi folosite, tranzacțiile implicând tabelele din schemele utilizatorilor care inițiază execuția. Prin urmare, în schema PERSONAL_MASTER vom executa mai întâi comenzile din listing 4.4 (tabelele), apoi, în ordine, cele din listing 9.34 (antetul PACHET_EXISTA), listing 9.28 (corpul PACHET_EXISTA), listing 9.35 (pachetul PACHET_SALARIZARE), listing 9.31 (corpul PACHET_SALARIZARE), listing 9.33 (procedura P_ACT_SP_SA2).

Pentru a verifica rezultatul final să comparăm rezultatul execuției următoarei fraze SELECT pentru schema PERSONAL inițială și pentru schema PERSONAL_MASTER:

```
SELECT object_name, authid FROM user_procedures;
```

```
SQL> SELECT object_name, authid FROM user_procedures;
```

OBJECT_NAME	AUTHID
PACHET_EXISTA	DEFINER
PACHET_EXISTA	DEFINER
PACHET_EXISTA	DEFINER
PACHET_SALARIZARE	DEFINER
PACHET_SALARIZARE	DEFINER
PACHET_SALARIZARE	DEFINER
PACHET_SALARIZARE	DEFINER
P_ACT_SP_SA2	DEFINER

Figura 9.30_1. În schema PERSONAL obiectele sunt create cu drepturi de execuție locale

```
SQL> SELECT object_name, authid FROM user_procedures;
```

OBJECT_NAME	AUTHID
PACHET_EXISTA	CURRENT_USER
PACHET_EXISTA	CURRENT_USER
PACHET_EXISTA	CURRENT_USER
PACHET_SALARIZARE	CURRENT_USER
PACHET_SALARIZARE	CURRENT_USER
PACHET_SALARIZARE	CURRENT_USER
PACHET_SALARIZARE	CURRENT_USER
P_ACT_SP_SA2	CURRENT_USER

Figura 9.30_2. În schema PERSONAL_MASTER obiectele sunt create cu drepturi de execuție pentru utilizatorul curent

În continuare, pentru a utiliza în mod partajabil procedura P_ACT_SP_SA2 vom acorda drepturi de execuție utilizatorilor PERSONAL și PERSONAL2 (mai vechile noastre cunoștințe).

```
GRANT EXECUTE ON p_act_sp_sa2 TO personal, personal_2
```

Ne întoarcem acum în sesiunea deschisă folosind contul PERSONAL_2 (sau deschidem una nouă) unde vom rula scriptul din listing 4.4 (crearea tabelelor PERSONAL, PONTAJE, RETINERI, SALARII, SPORURI, TRANSE_SV) după care vom popula tabelele PERSONAL, PONTAJE și TRANSE_SV. Din aceeași sesiune verificăm dacă PERSONAL_2 are acces la procedura P_ACT_SP_SA2 din schema PERSONAL_MASTER și, de asemenea, conținutul tabeli SPORURI. Presupunem că situația inițială în schema PERSONAL_2 ar fi cea din figura 9.31, unde se observă că, pentru a verifica obiectele accesibile nu numai din schema locală ci și din alte scheme, am interogât tabela virtuală ALL_OBJECTS și nu USER_OBJECTS.

```
SQL> SELECT object_name, owner FROM all_objects WHERE owner = 'PERSONAL_MASTER';
```

OBJECT_NAME	OWNER
P_ACT_SP_SA2	PERSONAL_MASTER

```
SQL> SELECT * FROM pontaje;
```

MARCA	DATA	ORELUCRATE	ORECO	ORENOAPTE	OREABSNEM
2001	07-FEB-03	6	0	0	0
2002	07-FEB-03	6	0	0	0
2003	07-FEB-03	6	0	0	0
2004	07-FEB-03	6	0	0	0
2002	10-FEB-03	10	0	2	0
2001	10-FEB-03	10	0	2	0
2004	11-FEB-03	4	0	4	0
2001	11-FEB-03	4	0	4	0

8 rows selected.

```
SQL> SELECT * FROM sporuri;
```

no rows selected

Figura 9.31. Situația inițială în schema PERSONAL_2

Executăm apoi procedura P_ACT_SP_SA2 din schema PERSONAL_MASTER și verificăm din nou „consistența” tabelului SPORURI (figura 9.32).

```
SQL> SELECT * FROM sporuri;
```

no rows selected

```
SQL> EXECUTE personal_master.p_act_sp_sa2(2003, 2);
```

PL/SQL procedure successfully completed.

```
SQL> SELECT * FROM sporuri;
```

MARCA	AN	LUNA	SPUECH	ORENOAPTE	SPNOAPTE	ALTESP
2001	2003	2	66000	6	30000	0
2002	2003	2	48000	2	13000	0
2003	2003	2	27000	0	0	0
2004	2003	2	105000	4	42000	0

Figura 9.32. Situația din schema PERSONAL_2 după execuția procedurii PERSONAL_MASTER.P_ACT_SP_SA2

Este evident că procedura din schema PERSONAL_MASTER s-a executat fără probleme, deși, la prima vedere, ne puteam îndoi de acest lucru din cel puțin două puncte de vedere:

- procedurile și funcțiile la care se fac referințe în corpul procedurii P_ACT_SP_SA2 nu se regăsesc în schema curentă din care a fost invocată (PERSONAL_2), ci în schema originală în care a fost creată;
- utilizatorului curent nu i s-au acordat drepturi de execuție explicite pentru aceste proceduri.

În acest sens, regula care se aplică este următoarea: referințele externe și drepturile asupra obiectelor implicate de aceste referințe în frazele SELECT, INSERT, UPDATE, DELETE, LOCK TABLE, OPEN, OPEN FOR sau EXECUTE IMMEDIATE și OPEN-FOR-USING (SQL dinamic) sunt rezolvate în schema utilizatorului curent, pe când toate celelalte referințe (în speță cele către obiecte procedurale) sunt verificate și rezolvate în schema utilizatorului proprietar al schemei care le deține, atât în momentul compilării cât și la execuție.

9.5.2. Variabile cursor

Variabilele obișnuite din blocurile PL/SQL punctează direct către locația în care se găsesc „valorile” referite. Din acest punct de vedere, am putea spune că „variabilă reprezintă valoarea”. Mai „vechii” în programarea orientată obiect știu însă că acest mod de a pune problema nu oferă nici un fel de flexibilitate când între blocuri de program distincte sunt schimbate (de cele mai multe ori, prin parametrizare) structuri complexe de date care se doresc partajabile. Mai exact, atunci când se transmite „valoarea” unei variabile, este transmis literalmente întreg „conținutul” acesteia. Ce se întâmplă însă în cazul în care conținutul unei variabile

reprezintă o colecție ceva mai voluminoasă de elemente, fiecare cu o „structură liniară” bogată ? Acesta este și cazul cursoroanelor PL/SQL: pentru a transmite conținutul unei astfel de structuri de date, va trebuie preluată fiecare înregistrare și transmisă individual în locația de destinație. Prin urmare, este mai eficient ca în locul "pasării" conținutul unui cursor, să se trimită o referință spre locația în care s-a format inițial rezultatul interogării suport. Aceasta este menirea sau „rațiunea de a fi” pentru *variabilele de tip cursor*.

O *variabilă cursor* reprezintă un „pointer” în care va fi stocată, la inițializare, o referință către un cursor Oracle. Prin urmare, odată inițializată o astfel de variabilă, cursorul țintă va putea fi parcurs în contextul (blocul anonim, procedura, funcția) în care a fost declarată variabila.

Care ar fi utilitatea unor astfel de variabile ? Meritul cel mai important al lor este că pot menține *dinamic* o referință către un cursor și, prin urmare, acesta nu trebuie declarat și deschis în procedura în care este plasată variabila, putând fi „împrumutat” din alt context (bloc, procedură, funcție), de cele mai multe ori prin mecanismul de parametrizare. Rezultatul imediat al unei astfel de „partajări” este reducerea traficului de date și consumului de resurse (memorie).

Înainte de utilizarea acestor variabile, tipul acestora trebuie declarat ca fiind o referință:

```
DECLARE
  -- tipul înregistrărilor cursorului
  TYPE rec_tip IS RECORD(marca personal.marca%TYPE,
    spor_vechime sporuri.spvech%TYPE);
  -- tipul ce reprezintă referința la un cursor
  TYPE ref_crs_tip IS REF CURSOR RETURN rec_tip;
  -- declararea variabilei
  v_ref_crs ref_crs_tip;
BEGIN
  --- ---
END;
```

Dacă tipul REF CURSOR este însoțit de specificarea tipului înregistrărilor returnate, atunci variabila declarată astfel va fi tipizată „puternic”, în caz contrar fiind tipizată „slab”. Importanța modului de tipizare este relevantă de probabilitatea apariției erorilor la execuție. Dacă variabila este tipizată „puternic”, atunci compilatorul PL/SQL poate face verificările de rigoare cu privire la operațiile la care este supusă, putând împiedica încă din „fașă” anumite inconsistențe care se pot dovedi fatale în momentul execuției (de exemplu, prelucrarea unei componente din structura de date a variabilei care nu se va materializa la execuție, nefiind consemnată în tipul returnat).

Pentru a exemplifica formarea unor astfel de variabile, partajarea cursoroanelor la care fac referire și parcurgerea acestora, vom crea o structură de proceduri prin care să putem afișa alternativ o situație centralizată fie despre salarii, fie despre sporuri, fie despre rețineri. Încercăm să evităm declararea a trei cursori diferite pentru fiecare dintre cele trei categorii de liste. În acest scop vom crea mai întâi un pachet în care să definim tipul REF CURSOR pe care îl apela ulterior.

Listing 9.36. Definirea tipului REF CURSOR

```

CREATE OR REPLACE PACKAGE pachet_var_cursor AS
  TYPE rec_crs_salarizare IS RECORD
    (marca personal.marca%TYPE,
     numepren personal.numepren%TYPE,
     luna salarii.luna%TYPE,
     sumatotala salarii.venitbaza%TYPE);
  TYPE crs_salarizare IS REF CURSOR RETURN rec_crs_salarizare;
END;
```

Afișarea celor trei situații amintite mai sus beneficiază de o procedură specială, AFISEAZA_SITUATIE_SALARII, al cărei principal rol este executarea instrucțiunilor DBMS_OUTPUT.PUT_LINE pentru fiecare salariat obținut dintr-o variabilă cursor al cărei tip l-am definit mai înainte (vezi listing 9.38). Vom parametriza această procedură după codul compartimentului, tipul listei de obținut (sporuri, salarii, rețineri), și intervalul lunar. Cursorul propriu-zis cu datele ce vor fi afișate de către procedura AFISEAZA_SITUATIE_SALARII este creat de către procedura DATE_SALARIZARE care primește ca parametri o parte din valorile de selecție (compartimentul, intervalul lunar, tipul listei), dar prezintă și un parametru IN OUT destinat referinței definite prin variabila cursor (vezi listing 9.37).

Listing 9.37. Procedura pentru inițializarea cursorului

```

CREATE OR REPLACE PROCEDURE date_salarizare
  (compartiment IN personal.compart%TYPE,
   tabela IN NUMBER,
   crs_sal IN OUT pachet_var_cursor.crs_salarizare,
   luna_i_perioada IN NUMBER,
   luna_sf_perioada IN NUMBER, anul IN NUMBER) IS
BEGIN
  IF tabela = 1 THEN
    OPEN crs_sal FOR
      SELECT p.marca, numepren, luna, (spvech + spnoapte + altesp) sumatotala
      FROM personal p, sporuri s
      WHERE p.marca = s.marca
        AND luna >= luna_i_perioada AND luna <= luna_sf_perioada
        AND an = anul AND p.compart = compartiment;
  ELSIF tabela = 2 THEN
    OPEN crs_sal FOR
      SELECT p.marca, numepren, luna, (popriri + car + alteret) sumatotala
      FROM personal p, retineri r
      WHERE p.marca = r.marca
        AND luna >= luna_i_perioada AND luna <= luna_sf_perioada
        AND an = anul AND p.compart = compartiment;
  ELSIF tabela = 3 THEN
    OPEN crs_sal FOR
      SELECT p.marca, numepren, luna, (venitbaza + sporuri - impozit - retineri)
        sumatotala
      FROM personal p, salarii s
      WHERE p.marca = s.marca
        AND luna >= luna_i_perioada AND luna <= luna_sf_perioada
        AND an = anul AND p.compart = compartiment;
  END IF;
END;
```

Listing 9.38. Procedura pentru afișarea listei obținute

```

CREATE OR REPLACE PROCEDURE afiseaza_situatie_salarii
(compart VARCHAR2,
tip_lista VARCHAR2,
prima_luna salarii.luna%TYPE,
ultima_luna salarii.luna%TYPE)
IS
    crs_sal pachet_var_cursor.crs_salarizare;
    rec_date pachet_var_cursor.rec_crs_salarizare;
    anul_curent NUMBER := TO_CHAR(SYSDATE, 'YYYY');
    tabela NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Lista ' || tip_lista || ' pentru compartimentul ' || UPPER(compart));

    IF UPPER(tip_lista) = 'SPORURI' THEN
        tabela := 1;
    ELSIF UPPER(tip_lista) = 'RETINERI' THEN
        tabela := 2;
    ELSIF UPPER(tip_lista) = 'SALARII' THEN
        tabela := 3;
    END IF;
    date_salarizare(UPPER(compart), tabela, crs_sal, prima_luna, ultima_luna, anul_curent);
    -- nu vom folosi WHILE (crs_sal%FOUND)
    LOOP
        FETCH crs_sal INTO rec_date;
        EXIT WHEN crs_sal%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Salariat: ' || rec_date.numepren);
        DBMS_OUTPUT.PUT_LINE('Luna: ' || rec_date.luna);
        DBMS_OUTPUT.PUT_LINE('Valoare ' || tip_lista || ': ' || rec_date.sumatotala);
    END LOOP;
    CLOSE crs_sal;
END;

```

Rolul principal îl joacă variabila `crs_sal` declarată în procedura `AFISEAZA_SITUATIE_SALARII`. Cursorul parcurs se formează prin apelul procedurii `DATE_SALARII` folosind o instrucțiune `OPEN-FOR`. Acest cursor a cărui referință inițializează variabila `crs_sal`, este parcurs ca unul obișnuit printr-o instrucțiune `FETCH` și va fi activ cât timp va exista o variabilă care să facă referire la el, sau până când este închis explicit printr-o instrucțiune `CLOSE`. Rezultatul execuției procedurii `AFISEAZA_SITUATIE_SALARII` ar putea fi cel din figura 9.33.

```

SQL> execute afiseaza_situatie_salarii('conta', 'salarii', 1, 3)
Lista salarii pentru compartimentul CONTA
Salariat: Angajat 1
Luna: 3
Sporuri: 385000
Salariat: Angajat 2
Luna: 3
Sporuri: 870000

```

PL/SQL procedure successfully completed.

Figura 9.33, Rezultatul parcurgerii cursorului referit printr-o variabilă locală a procedurii AFISEAZA_SITUATIE_SALARII

9.5.3. Nivele de puritate ale funcțiilor apelabile în fraze SELECT

În capitolul 9 s-au prezenta câteva exemple de invocare a unor funcții utilizator în fraze SELECT. De exemplu, pentru a lista salariații din tabela PERSONAL astfel încât `datasv` să fie evaluată în ani vechime, putem apela la funcția `F_ANI_VECHIME` din pachetul `PACHET_SALARIZARE` invocând-o într-o frază SELECT cum ar fi:

```
SELECT marca, numepren,
       pachet_salarizare.f_an_ivechime(datasv,
       TO_CHAR(SYSDATE, 'YYYY'), TO_CHAR(SYSDATE, 'MM')) vechimea
FROM personal
```

Rezultatul execuției acestei fraze SELECT ar putea fi cel din figura 9.34.

Invocarea funcției din fraza SELECT exemplificată anterior a decurs fără probleme (ca dovadă rezultatul), însă nu întotdeauna lucrurile ar putea fi atât de „roz”. Cu alte cuvinte, pentru ca o astfel de frază SELECT să poată fi executată, funcțiile implicate trebuie să respecte anumite restricții:

```
SQL> SELECT marca, numepren, pachet_salarizare.f_an_ivechime(datasv,
2    TO_CHAR(SYSDATE, 'YYYY'), TO_CHAR(SYSDATE, 'MM')) vechimea FROM personal
3    /
```

MARCA	NUMEPREN	VECHIMEA
1001	Angajat 1	7
1002	Angajat 2	4
1003	Angajat 3	10
1004	Angajat 4	2

Figura 9.34. Invocarea unei funcții stocate dintr-o frază SELECT-SQL

- în primul rând, o funcție apelabilă dintr-o interogare sau frază DML nu trebuie să încheie tranzacția curentă (ROLLBACK sau COMMIT), nu trebuie să creeze, sau să facă un ROLLBACK la un *savepoint*, nu trebuie să modifice (ALTER) starea sistemului sau sesiunii curente;
- în al doilea rând, o funcție apelabilă dintr-o interogare (SELECT) nu trebuie să încerce să modifice baza de date printr-o frază DML sau prin oricare altă modalitate (nivelul de puritate WNDS, vezi mai jos);
- în al treilea rând, o funcție apelabilă dintr-o frază DML nu trebuie să consulte sau să modifice tabela specifică care reprezintă obiectivul acelei fraze SQL.

De asemenea, în legătură cu modul de redactare a unei funcții apelabile într-o expresie SQL, mai trebuie respectate următoarele condiții:

- trebuie să fie vorba despre o funcție stocată, nu despre o funcție definită și apelată în cadrul unui bloc anonim;
- funcția respectivă poate prelua doar valori scalare sau de tip înregistrare, nu colecții tabelare;
- toți parametrii formali ai funcției trebuie să fie de tip `IN`;
- tipurile de date atât pentru parametrii formali cât și pentru valoarea returnată trebuie să fie tipuri interne Oracle (`CHAR`, `DATE`, `NUMBER` etc.) nu tipuri PL/SQL specifice, cum sunt `BOOLEAN`, `RECORD` sau `TABLE`;

Controlul asupra efectelor secundare pe care invocarea unei funcții într-o frază SQL le-ar putea avea asupra tabelor sau stării variabilelor împachetate este posibil prin intermediul nivelelor de puritate. Acestea pot preveni paralelizarea interogărilor și producerea unor rezultate dependente de ordinea de execuție, ceea ce ar induce un anumit grad de indeterminare asupra acestora. De asemenea, se poate solicita ca starea pachetelor să fie consistentă (menținută nemodificată) în decursul sesiunilor utilizatorilor. Prin urmare, funcție de *nivelul de puritate*, funcțiile pot fi constrânse să respecte următoarele restricții:

- nemodificarea bazei de date – nivelul `WNDS` (*Write No Database Statement*);
- nemodificarea stării variabilelor împachetate – nivelul `WNPS` (*Write No Package Statement*);
- neconsultarea datelor stocate în baza de date – nivelul `RNDS` (*Read No Database Statement*);
- neconsultarea stării variabilelor împachetate – nivelul `RNPS` (*Read No Package Statement*);

Aplicarea acestor nivele de puritate se face (de regulă în Oracle9i) la execuție funcție de contextul în care a fost invocată respectiva funcție. Spre exemplu, dacă vom crea o funcție asemănătoare cu `F_ANI_VECHIME` din `PACHET_SALARI-ZARE` dar, pe lângă calculul obișnuit, vom face și ceva modificări în anumite tabele, atunci, la execuție, serverul ne va indica „infracțiunea” comisă.

Listing 9.39 Crearea unei funcții de test pentru nivelul de puritate

```
CREATE TABLE vechime_tmp (datasv DATE, data_calcul DATE, ani_vechime NUMBER(2))
/

CREATE OR REPLACE FUNCTION f_vechime (datasv_personal.datasv%TYPE,
    an_IN_salarii.an%TYPE, luna_salarii.luna%TYPE
) RETURN transe_sv.ani_limita_inf%TYPE
AS
    prima_zi DATE := TO_DATE('01/'||luna_||'|'||an_, 'DD/MM/YYYY') ;
BEGIN
```



```

INSERT INTO vechime_tmp VALUES (datasv_, prima_zi,
    TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0));
RETURN TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0);
END f_vechime;

```

Crearea funcției F_VECHIME prin rularea listingului anterior nu produce nici o eroare de compilare, însă invocarea ei într-un SELECT, produce rezultatul (eroarea) din figura 9.35.

```

SQL> SELECT marca, numepren, f_vechime(datasv,
2     TO_CHAR(SYSDATE, 'YYYY'), TO_CHAR(SYSDATE, 'MM')) vechimea FROM personal
3 /
SELECT marca, numepren, f_vechime(datasv,
*
ERROR at line 1:
ORA-14551: cannot perform a DML operation inside a query

```

Figura 9.35. Invocarea unei funcții „indecente” într-o frază SELECT-SQL

Verificarea nivelului de puritate în momentul compilării, pentru evitarea erorilor la execuție de genul celei din figura 9.35, poate fi realizată folosind directiva PRAGMA RESTRICT_REFERENCE. Această directivă este obligatorie în versiunile 8i și anterioare, însă mecanismul nivelului de puritate a fost relaxat în 9i prin posibilitatea efectuării verificărilor în momentul execuției față de momentul compilării. Cu alte cuvinte, „nevinovata” funcție PACHET_SALARIZARE.F_ANI_VECHIME, implicată în fraza SELECT din figura 9.34, nu ar putea fi invocată în acest fel într-o bază de date 8i, obligați fiind să determinăm verificăm nivelului ei de puritate la crearea/compilarea ei printr-o directivă specifică. În versiunile 9i, utilizarea directivei de compilare RESTRICT_REFERENCE are ca efect anularea activării mecanismului de verificare a nivelului de puritate la momentul execuției. Sintaxa acestei directive este următoarea:

```

PRAGMA RESTRICT_REFERENCE (subprogram_sau_nume_package, WNDS
    [, WNPS [, RNDS] [, RNPS]);

```

Această instrucțiune va însoți declarațiile procedurilor în antetul pachetelor din care fac parte. În acest sens, procedurile F_ANI_VECHIME și F_PROCENT_SPOR_VECHIME ar putea fi urmate de către o directivă RESTRICT_REFERENCE, care să asigure invocarea lor independentă în fraze SELECT-SQL (vezi listing 9.40).

Listing 9.40. Declarațiile din PACHET_SALARIZARE privind directivele RESTRICT_REFERENCE

```

CREATE OR REPLACE PACKAGE pachet_salarizare AS
---
--- declarațiile elementelor anterioare rămân neschimbate
---

```

```

FUNCTION f_ani_vechime (datasv_personal.datasv%TYPE,
    an_ IN salarii.an%TYPE, luna_salarii.luna%TYPE
) RETURN transe_sv.ani_limita_inf%TYPE ;
PRAGMA RESTRICT_REFERENCES(f_ani_vechime, WNDS, WNPS, RNPS);

FUNCTION f_procent_spor_vechime (ani_transe_sv.ani_limita_inf%TYPE)
    RETURN transe_sv.procent_sv%TYPE ;
PRAGMA RESTRICT_REFERENCES(f_procent_spor_vechime, WNDS, WNPS, RNPS);

---
--- declarațiile elementelor următoare rămân neschimbate
---

END pachet_salarizare ;

```

Pentru a evita repetarea instrucțiunii **PRAGMA RESTRICT_REFERENCE** în condițiile în care funcțiile **F_ANI_VECHIME** și **F_PROCENT_SPOR_VECHIME** sunt ultimele din antetul pachetului, atunci, înainte de declarația primei dintre ele, se poate folosi cuvântul cheie **DEFAULT**:

```

PRAGMA RESTRICT_REFERENCES(DEFAULT, WNDS, WNPS, RNPS);
FUNCTION f_ani_vechime
    (datasv_personal.datasv%TYPE, ... ..
    ... ..
FUNCTION f_procent_spor_vechime
    (ani_transe_sv.ani_limita_inf%TYPE)
    ... ..

```

Dacă însă vom încerca să modificăm, de exemplu, funcția **F_ANI_VECHIME** în sensul consultării unor variabile din pachetul **PACHET_SALARIZARE** și introducerea unei comenzi **DML** (vezi listing 9.41), vom avea parte de o eroare de compilare (vezi figura 9.36) ca urmare a nerespectării indicației de ne-consultare a variabilelor împachetate (**RNPS**) și ne-modificare a stării bazei de date (**WNDS**).

Listing 9.41. Modificarea funcției **F_ANI_VECHIME** pentru a „viola” directiva specificată în antetul pachetului

```

CREATE OR REPLACE PACKAGE BODY pachet_salarizare AS

---
--- definițiile elementelor anterioare rămân neschimbate
---
-----

FUNCTION f_ani_vechime (datasv_personal.datasv%TYPE,
    an_ IN salarii.an%TYPE ,
    luna_salarii.luna%TYPE
) RETURN transe_sv.ani_limita_inf%TYPE
AS
    an__ salarii.an%TYPE := NVL(an_ , pachet_salarizare.v_an);
    luna__ salarii.an%TYPE := NVL(luna_ , pachet_salarizare.v_luna);
    prima_zi DATE := TO_DATE('01/'||luna__||'/'||an__, 'DD/MM/YYYY') ;
BEGIN

```

```

INSERT INTO vechime_tmp(datasv_, prima_zi,
TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0));

RETURN TRUNC(MONTHS_BETWEEN(prima_zi, datasv_) / 12,0);
END f_anii_vechime;

-----
FUNCTION f_procent_spor_vechime (ani_transe_sv.ani_limita_inf%TYPE)
RETURN transe_sv.procent_sv%TYPE
AS
v_procent transe_sv.procent_sv%TYPE := 0;
BEGIN
-- ...v_transe_sv e cu siguranta initializat ..

-- determinarea procentului
FOR i IN 1..v_transe_sv.COUNT LOOP
    IF ani_ >= v_transe_sv(i).ani_limita_inf AND
        ani_ < v_transe_sv(i).ani_limita_sup THEN
        v_procent := v_transe_sv(i).procent_sv;
    EXIT;
END IF;
END LOOP;
RETURN v_procent;
END f_procent_spor_vechime;

-----

---
--- definițiile elementelor următoare rămân neschimbate
---
END pachet_salarizare;

```

```

68 END pachet_salarizare;
69 /

```

Warning: Package Body created with compilation errors.

```

SQL> show errors
Errors for PACKAGE BODY PACHET_SALARIZARE:

```

LINE/COL ERROR

```

-----
32/1      PLS-00452: Subprogram 'F_ANI_UECHIME' violates its associated
          pragma

45/1      PLS-00452: Subprogram 'F_PROCENT_SPOR_UECHIME' violates its
          associated pragma

```

Figura 9.36. Erorile de compilare produse de „violarea” condițiilor impuse de pragmele din antetul pachetului PACHET_SALARIZARE

Se observă că și F_PROCENT_SPOR_UECHIME are ceva probleme datorită faptului că determinarea sporului de vechime se face prin parcurgerea vectorului asociativ v_transe_sv, nerespectând astfel nivelul de puritate RNDS. Cuvântul cheie TRUST ne-ar permite să „înșelăm” compilatorul în privința declarațiilor

noastre din directivele `RESTRICT_REFERENCE`. Astfel, am putea modifica `PRAGMA RESTRICT_REFERENCE` din listing 9.40 așa cum arată în listingul 9.42.

Listing 9.42. Declarațiile din `PACHET_SALARIZARE` privind directivele `RESTRICT_REFERENCE`

```
CREATE OR REPLACE PACKAGE pachet_salarizare AS
---
--- declarațiile elementelor anterioare rămân neschimbate
---

FUNCTION f_an_i_vechime (datasv_personal.datasv%TYPE,
    an_IN_salarii.an%TYPE, luna_salarii.luna%TYPE
) RETURN transe_sv.an_i_limita_inf%TYPE ;
PRAGMA RESTRICT_REFERENCES(f_an_i_vechime, WNDS, WNPS, RNPS, TRUST);

FUNCTION f_procent_spor_vechime (an_i_transe_sv.an_i_limita_inf%TYPE)
    RETURN transe_sv.procent_sv%TYPE ;
PRAGMA RESTRICT_REFERENCES(f_procent_spor_vechime, WNDS, WNPS, RNPS, TRUST);

---
--- declarațiile elementelor următoare rămân neschimbate
---

END pachet_salarizare ;
```

Prin această modificare execuția comenzii de crearea a corpului `PACHET_SALARIZARE` din listing 9.41 se va derula fără probleme. Acest lucru nu împiedică verificarea nivelului de puritate `WNDS` la execuție atunci când vom încerca implicarea funcției `F_ANI_VECHIME` în fraza `SELECT` de la începutul acestui paragraf:

```
SELECT marca, numepren,
    pachet_salarizare.f_an_i_vechime(datasv,
        TO_CHAR(SYSDATE, 'YYYY'),
        TO_CHAR(SYSDATE, 'MM')) vechimea
FROM personal
```

```
SQL> SELECT marca, numepren, pachet_salarizare.f_an_i_vechime(datasv,
2   TO_CHAR(SYSDATE, 'YYYY'), TO_CHAR(SYSDATE, 'MM')) vechimea FROM personal
3 /
SELECT marca, numepren, pachet_salarizare.f_an_i_vechime(datasv,
    *
```

ERROR at line 1:
ORA-14551: cannot perform a DML operation inside a query

Figura 9.37 Verificarea la execuție a nivelului de puritate `WNDS` pentru `F_ANI_VECHIME`