Stored Procedures

Abstracts from PostgreSQL documentation

pm jat @ daiict

Stored Procedures

- What is it; understand “stored” and “procedure”?
- Benefits?
- When do you need them?
- How do you create and use?

Stored Procedures – motivation

- Consider a scenario of DA-IICT inviting applications for B.Tech. admissions.
- Let us say we have a relation as following- Applications(AppNo, Name, JEE_Marks_Phy, JEE_Marks_Chem, JEE_Marks_Math, DA_Rank)
- We want to allocate DA_Rank to each applicant. Can you write SQL query for the this requirement?
- To make if more complex, let us say, we need to allocate Cat_Rank as well?
- SQL is just not enough for such requirements?

Database Programming scenarios

- Again consider DA-IICT inviting applications-
  - **Student Applies**: showing up blank application form may too require getting various inputs from databases, applicant enters his/her details
  - On submission, various validation happens, may be by looking into databases
  - Finally new tuples (for new applicant) are inserted into relevant relations
  - DA-IICT prepares its on merit list; a process that updates few fields of a applicant relation.
  - DA-IICT allocates seats as per available sheet matrix (program wise, category wise, etc); updates again some fields of some relations

Database Programming scenarios

- In some data access (read/write) cases SQL is enough.
- In some cases you need iterations, for example category wise rank allocation (example below).
- There are many contexts in which this functionality can be performed: as stored procedure, in host languages by making appropriate API calls.

```
resultset = conn.execute("SELECT * FROM Merit_SC“)
//can assume that the query gives you shorted tuples in descending order of marks
rank = 0
While (!rs.EOF()) {
      rank = rank + 1
      con.Execute("update applicants set rank_cat = " + rank
                + " where application_no = " + resultset.application_no
    resultset.moveNext()
}
```

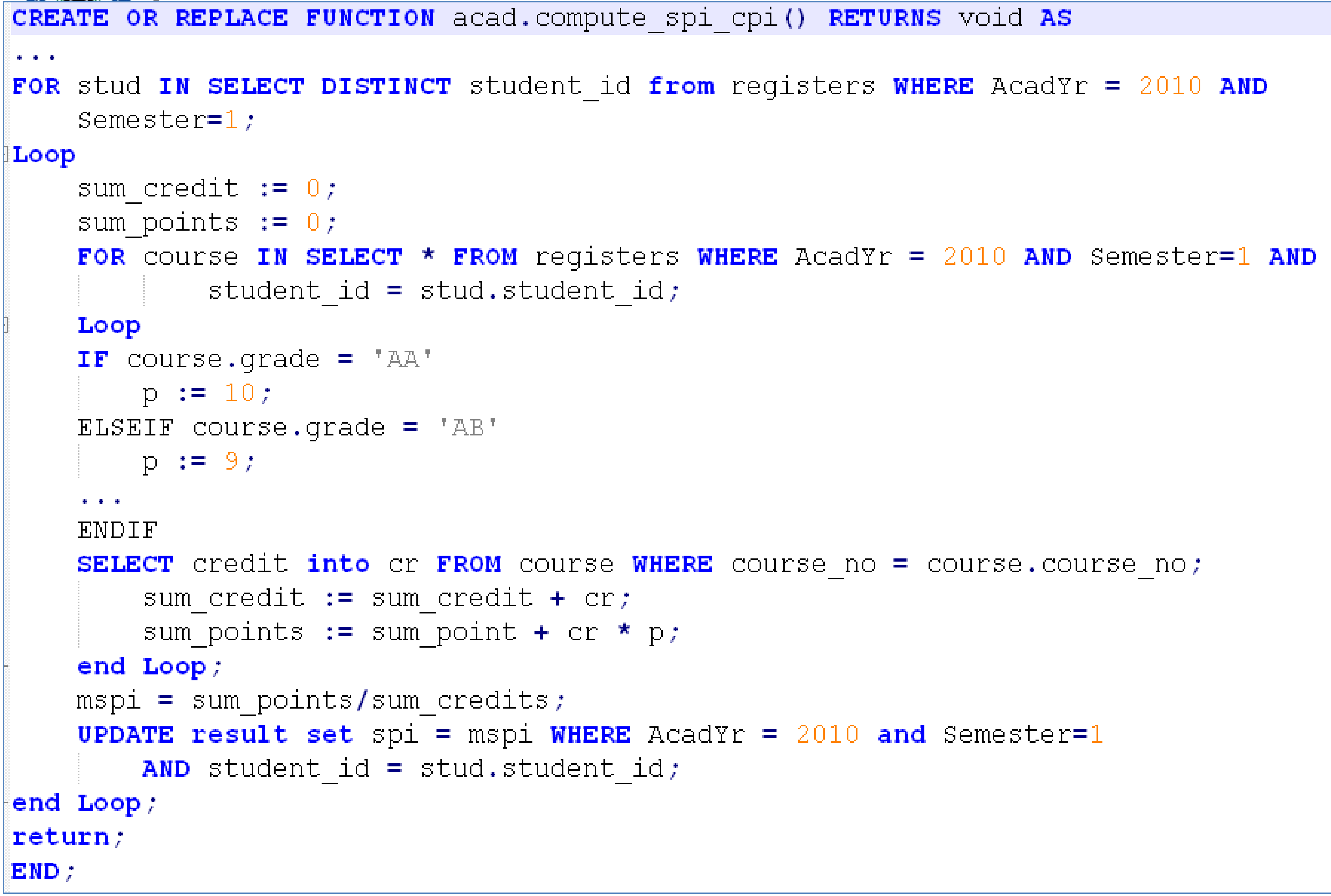
Database Programming Models

- SQL queries and Views
- **Stored Procedures**: programs that are stored as part of database schema
- General programming language programs accessing databases through API like JDBC (or ODBC/OLE-DB)
- **Embedded SQL**: SQL is permitted in programming – Oracle's Pro*C, PostgreSQL's ECPG, and SQLJ are such environments
- **Native Programming Interface**: Oracle Call Interface (OCI) and libpq with PostgreSQL allow you to communicate with databases directly
- Understand for each of above: context of run, experience writing simple programs [exams: may be asked to write pseudo code]

Stored Procedure – What is it?

- Stored procedure too are meant to manipulate databases, but way more than SQL.
- There are many complex manipulation operations that can not be done using SQL (Examples?)
- We can very well call them “stored functions”.
- Stored procedure is program function with following additional characteristics-
  - Stored as schema element
  - Is a Runs in “DBMS instance context”
  - Has access to all schema elements

Stored Procedure Example



```
CREATE OR REPLACE FUNCTION acad.compute_spi_cpi() RETURNS void AS
...
FOR stud IN SELECT DISTINCT student_id from registers WHERE AcadYr = 2010 AND
    Semester=1;
Loop
    sum_credit := 0;
    sum_points := 0;
    FOR course IN SELECT * FROM registers WHERE AcadYr = 2010 AND Semester=1 AND
            student_id = stud.student_id;
    Loop
    IF course.grade = 'AA'
        p := 10;
    ELSEIF course.grade = 'AB'
        p := 9;
    ...
    ENDIF
    SELECT credit into cr FROM course WHERE course_no = course.course_no;
        sum_credit := sum_credit + cr;
        sum_points := sum_point + cr * p;
    end Loop;
    mspi = sum_points/sum_credits;
    UPDATE result set spi = mspi WHERE AcadYr = 2010 and Semester=1
        AND student_id = stud.student_id;
end Loop;
return;
END;
```

Stored Procedure – benefits/applications

- Programs runs in immediate vicinity of data, and does not require any transportation of data over networks, and this
  - Reduce network overhead
  - Speeds up data processing
- Helps in enabling various operational abstractions over data
- Database triggers are implemented as stored procedures
- Complex constraints are also enforced using stored procedures.
- Also provides a mechanism of sharing a function/procedure by multiple applications
- PS: not all database processing may be implemented as stored procedure?

Examples of Stored Procedure

- Get grades of a student for a given semester.
- Compute end of the day and end of month procedures in an accounting database.
- Compute raise for employees

Language for Stored Procedures

- SQL is not enough?
- Stored procedures are written for requirements that can not be expressed in expressive queries like SQL, and **require procedural constructs** like branching and iterations.
- Note that newer releases of SQL do support some branching expressive power in SQL queries (IF, SWITCH or so)
- ANSI added as SQL/PSM (SQL/Persistent Stored Module) as part of SQL extension with SQL-1999.
- RDBMS like Oracle had PL/SQL even before this standardization.
- Most RDBMS, today, provide their own procedural languages for creating stored procedures. For example: PL/SQL (Oracle), SQL/PL (DB2), TSQL(MS SQL)
- These languages are also called SQL procedural languages. Remember these are extension to SQL

Stored Procedures in PostgreSQL

- PostgreSQL provides an open architecture for adding programming language to database server.
  - Requires you to load the language before programming in that language.
- Currently it supports C, PL/PgSQL, PL/perl, PL/Python, PL/Tcl, and PL/Java
- It also allows you to write simple stored procedures in SQL.
- Note: in most modern languages procedure are called as functions

Learning Stored Procedure Language (SPL)

- To repeat: Stored Procedure Language (SPL) is procedural extension to SQL
- Therefore, Let us look at following two aspects to it –
  - See what are "additional things" (over SQL) are required in a language to be used for writing stored procedures.
  - How do stored procedure differ with a procedure in a programming language like C

SPL = SQL + What ?

- Should allow to create and use variables
- Should allow to mix variables with attribute-names in SQL. SQL allows using only attribute names.
- Allow to submit a SQL statement to the database, and collect the responses in variables so that can be manipulated further
- Support for various procedure constructs like if .. then .. else, loops, exception handling, etc..

Creating and Using
Stored Procedures in PostgreSQL

- Language: let us look into PL/PgSQL
- **Create**:
  - Use CREATE OR REPLACE FUNCTION command. You define function here, almost in the same way we do in any programming language.
  - Function has Header and Body (contains functionality)
  - Function has Parameters, has return (with their types)
- **Execute**:
  - Invoke the function by sending appropriate values for input parameters.

PL/PgSQL summary

- Data Types:
  - all PostgreSQL data types are available
- Function Parameters
  - IN mode, INOUT mode, OUT mode
- Types for Parameter and Return-
  - all PostgreSQL data types, Record , Table-Row, and SET of Record/Row
- It is case insensitive, and strongly typed language

PL/PgSQL summary

- While implementing a stored function, you have access to database schema elements
- Language constructs like, If .. Else .. End if, case, FOR, WHILE loops etc..
- Cursors: enables navigating around result of a query
- Exceptions Handling
- Creating Functions, Triggers ..

PL/PgSQL Should be easier to learn?

BTW, How hard do you find understanding meaning of following code yourself ?

```
CREATE OR REPLACE FUNCTION valid_work_dept(ppno
  integer, pssn numeric) RETURNS boolean AS
  $BODY$
DECLARE
  edno integer;
BEGIN
  SELECT count(ssn) into ec FROM employee e,
  project p WHERE e.ssn = pssn AND e.dno = p.dno
  AND p.pno = ppno;
  if ec = 0 then
   return false;
  else
   return true;
  end if;
END
$BODY$ LANGUAGE plpgsql;
```

Variable Declarations

- The general syntax of a variable declaration is:

  name [CONSTANT] type [NOT NULL] [{ DEFAULT | := } expression];

- Below are some examples-

  user_id **integer**;

  quantity **numeric**(5);

  url **varchar(50)**;

  myrow **tablename%ROWTYPE**;

  myfield **tablename.columnname%TYPE**;

  arow **RECORD**;

Example

- A simple PL/pgSQL function – observe how variables are created and mixed with SQL statements

```
CREATE FUNCTION sales_tax(subtotal real, sale_type varchar(5)) RETURNS real
AS $BODY$
DECLARE
          tax_rate numeric(5,2);
BEGIN
          SELECT st.tax_rate into tax_rate FROM SALES_TYPE st
                   WHERE st.sale_type = sale_type
          RETURN subtotal * tax_rate / 100;
END;
$BODY$ LANGUAGE plpgsql;
```

- Call the function:

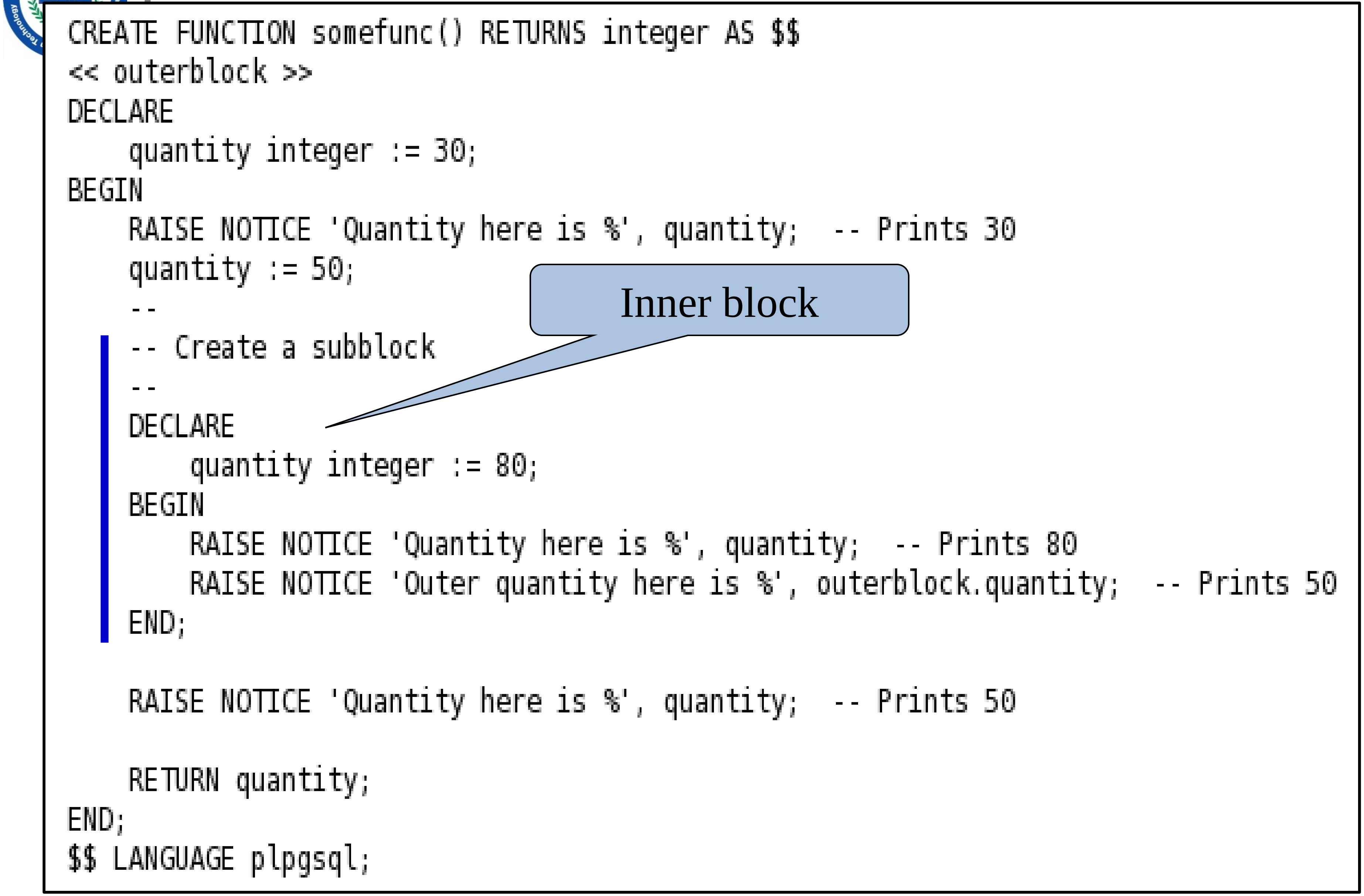
```
SELECT sales_tax(2250, 'C2');
```

Output: **3456.56**

PL/pgSQL function – function anatomy

```
CREATE FUNCTION sales_tax(subtotal real,
   sales_type varchar(5))-parameter variables

   RETURNS real - function header
   AS $BODY$ - implementation BEGINs

DECLARE –- local variables declared

   tax_rate numeric(5,2);

BEGIN
   SELECT st.tax_rate into tax_rate
      FROM sales_type st
         WHERE st.sales_type = sale_type
   RETURN subtotal * tax_rate / 100;

END;

$BODY$ –- implementation ENDs
   LANGUAGE plpgsql;
```

PL/pgSQL function – function anatomy



```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Inner block

Parameters to PL/pgSQL functions

- Parameters to PL/pgSQL functions are following types -
  - IN (default) – remains as constant within the function implementation
  - INOUT: carries value in and returns
  - OUT: return, can not carry in value
- PostgreSQL calls procedures to functions that have INOUT or OUT parameters.
- Example: Next

Example: Parameters to PL/pgSQL functions

```
CREATE OR REPLACE FUNCTION test12(IN x int, INOUT y
  int, OUT Z int) as $$

BEGIN

  y = y + x;

  z = y - x;

END $$ LANGUAGE 'plpgsql';
```

- Procedure called: SELECT test12(5, 6);
- Output: {11, 6}

Example: Functions returning values

```
CREATE OR REPLACE FUNCTION test13(x int, y int)
  RETURNS integer as $$
DECLARE
  z real;
BEGIN
  z = x*x - y*y;
  return z;
END $$ LANGUAGE 'plpgsql';
```

- Procedure called: SELECT test13(8, 6);
- Output: 28

Three ways of Executing SQL statements

- SELECT … INTO …
  - Used to collect the result into a variable, for example:
    `SELECT ssn FROM employee into essn;`
  - Note: in PL/pgSQL, you can not write
    `SELECT ssn FROM employee`
- PERFORM
  - Used when result of SQL statement is to be discarded. Normally used for executing a procedure that does not return anything or want to ignore to return, for example:
    `PERFORM query;`
- EXECUTE
  - Used for executing “dynamic SQL statements”; example
    `EXECUTE sql_string;`

Fetching query results into host variables

```
CREATE OR REPLACE FUNCTION valid_work_dept(ppno
  integer, pssn numeric) RETURNS boolean AS
  $BODY$
DECLARE
  edno integer;
BEGIN
  SELECT count(ssn) into ec FROM employee e,
  project p WHERE e.ssn = pssn AND e.dno = p.dno
  AND p.pno = ppno;
  if ec = 0 then
   return false;
  else
   return true;
  end if;
END
$BODY$ LANGUAGE plpgsql;
```

Fetching query results into host variables

```
CREATE OR REPLACE FUNCTION valid_work_dept(ppno int4, pssn numeric)
    RETURNS bool AS $$
DECLARE
    edno employee.dno%TYPE;
    pdno project.dno%TYPE;
BEGIN
    SELECT dno into edno FROM employee WHERE ssn = pssn;
    SELECT dno into pdno FROM project WHERE p.pno = ppno;
    if pdno = edno then
            return true; else return false;
    end if;
END $$ LANGUAGE 'plpgsql';
```

Dynamic Query in PL/pgSQL

- Query is built at run-time. For example consider following PL/pgSQL code fragments

```
query = 'UPDATE EMPLOYEE SET salary = salary + salary * ' ||
percent || ' WHERE dno = ' || mdno;
```

- Where query, mdno, and percent are host variables.

```
EXECUTE query;
```

Functions returning records

```
CREATE OR REPLACE FUNCTION test14(x int,  y
  int, OUT a int, OUT b int)
  RETURNS record as $$
BEGIN
  a = x + y;
  b = x * y;
END $$ LANGUAGE 'plpgsql';
```

- Procedure called: **`SELECT test14(5, 6);`**
  Outputs: **`{11, 6}`**
- Procedure called: **`SELECT test14(5, 6).a;`**
  Outputs: **`11`**

Functions returning RECORD

- Functions having output parameters, either you do not specify the return type, or
- Specify it to of type RECORD, as all out parameters are returned as a record
- Note the earlier function call:
  `SELECT (test12(5, 6)).y;`
  will output: **`11`**

Looping thru query results

```
CREATE or replace FUNCTION test_emp(pdno integer)
    RETURNS integer AS $$
DECLARE
    r record;
BEGIN
    FOR r IN SELECT * FROM employee WHERE dno = pdno
    LOOP
        --do whatever you want to do
        raise notice 'Name: %', r.fname;
    END LOOP;
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Functions returning ROW-SET

```
CREATE OR REPLACE FUNCTION empset()
       RETURNS SETOF employee AS $$
DECLARE
   e employee%rowtype;
BEGIN
   FOR e IN SELECT * FROM employee
   LOOP
       IF e.salary > 50000 THEN
              -- could be more complex filter
              RETURN NEXT e;
       END IF;
   END LOOP;
   RETURN;
END $BODY$ LANGUAGE 'plpgsql';
```

- Call: `SELECT * FROM empset();`
- Note: `* FROM` used only when functions returns a set of rows or records

Trapping Errors

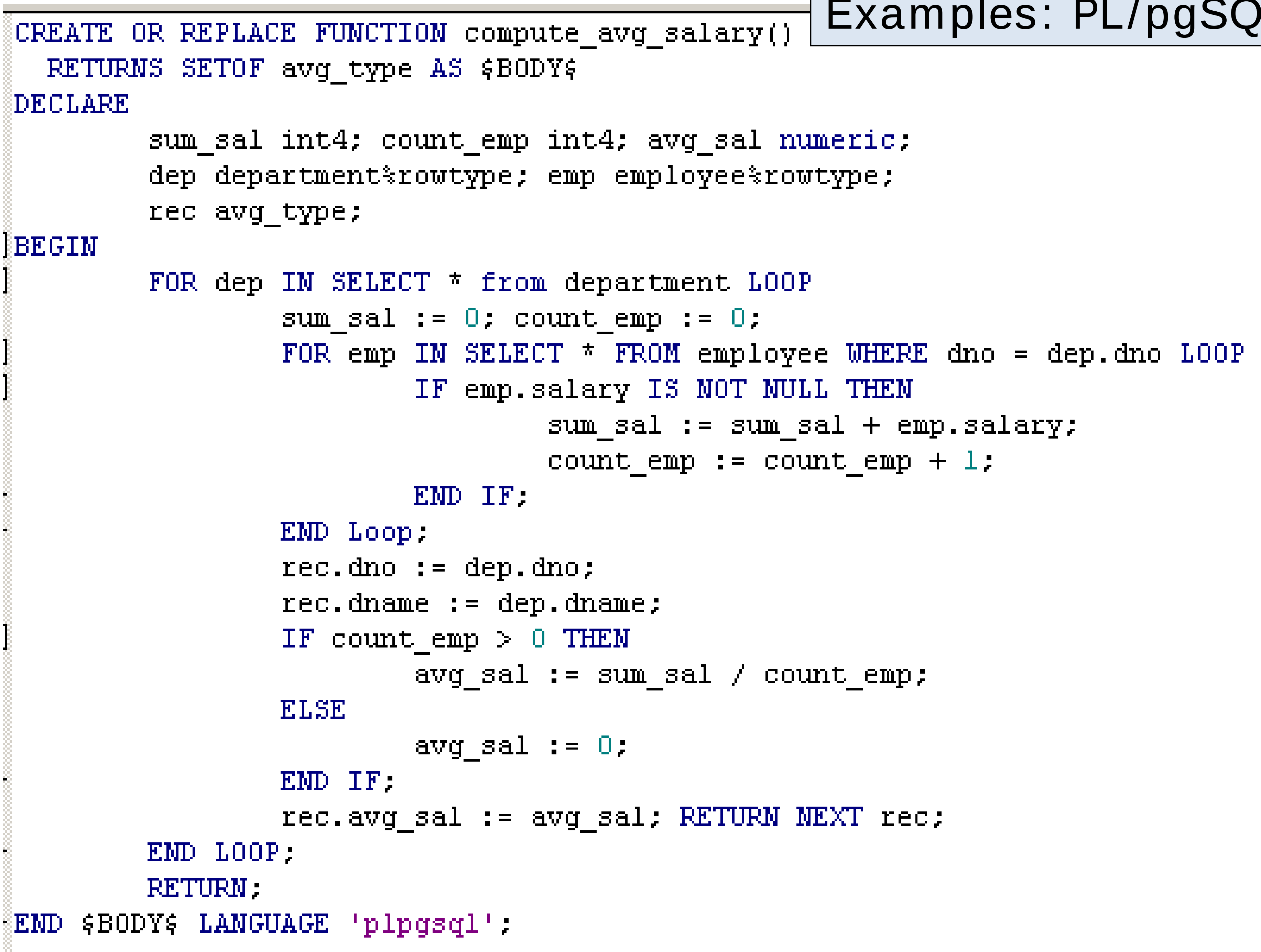
Analogous to

```
try {
  statements
}
catch (cond1 ) {
  handler_statements_1
}
catch (cond2 ) {
  handler_statements_2
}
```

```
BEGIN
   statements
   EXCEPTION
        WHEN cond1 THEN
                handler_statements_1
        [WHEN cond2 THEN
                handler_statements_2
        ... ]
END;
```

Here is complete list of exceptions: //intranet.daiict.ac.in/~pm_jat/postgres/html/errcodes-appendix.html

Examples: PL/pgSQL



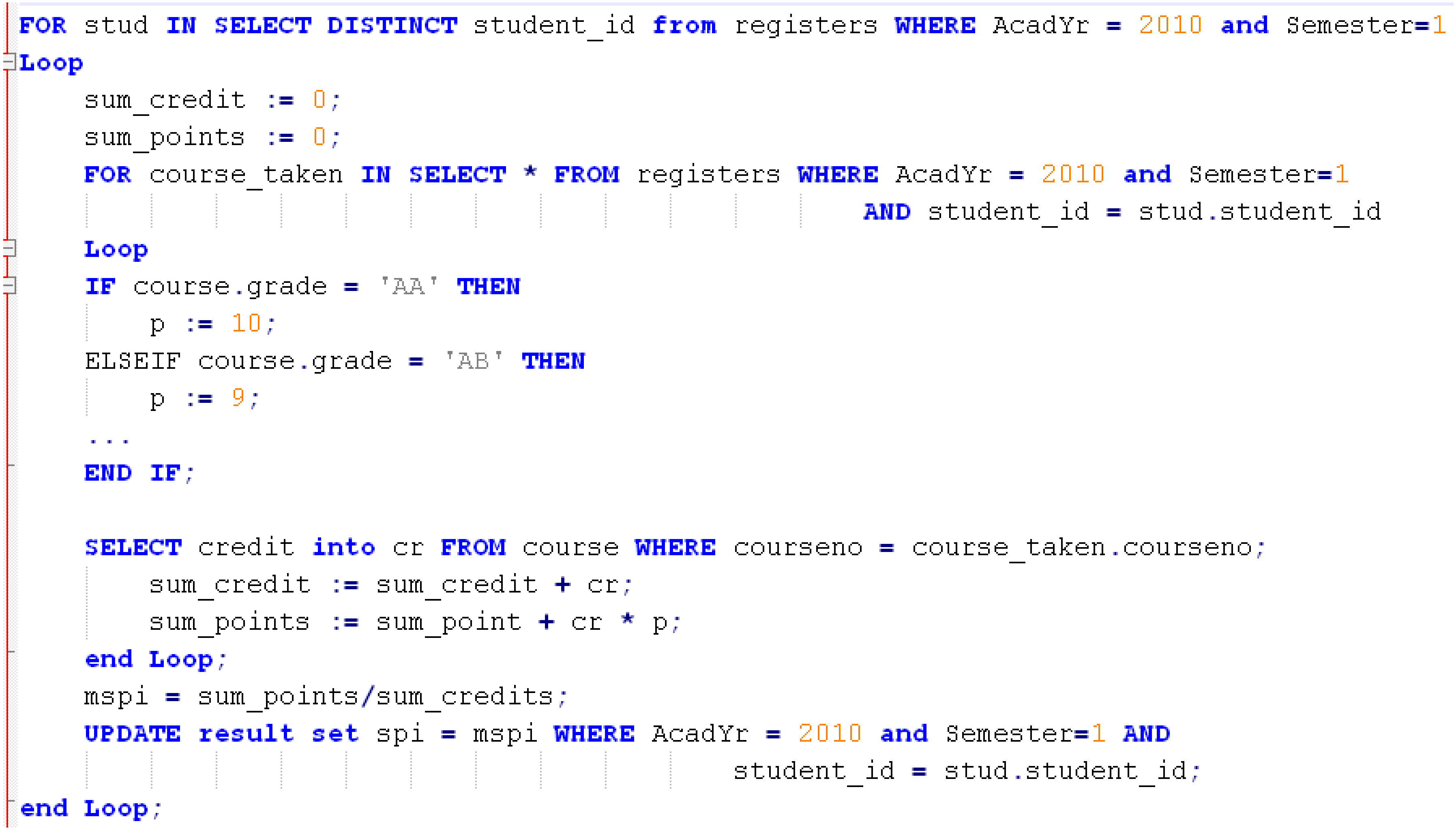
```
CREATE OR REPLACE FUNCTION compute_avg_salary()
 RETURNS SETOF avg_type AS $BODY$
DECLARE
    sum_sal int4; count_emp int4; avg_sal numeric;
    dep department%rowtype; emp employee%rowtype;
    rec avg_type;
BEGIN
    FOR dep IN SELECT * from department LOOP
        sum_sal := 0; count_emp := 0;
        FOR emp IN SELECT * FROM employee WHERE dno = dep.dno LOOP
            IF emp.salary IS NOT NULL THEN
                sum_sal := sum_sal + emp.salary;
                count_emp := count_emp + 1;
            END IF;
        END Loop;
        rec.dno := dep.dno;
        rec.dname := dep.dname;
        IF count_emp > 0 THEN
            avg_sal := sum_sal / count_emp;
        ELSE
            avg_sal := 0;
        END IF;
        rec.avg_sal := avg_sal; RETURN NEXT rec;
    END LOOP;
    RETURN;
END $BODY$ LANGUAGE 'plpgsql';
```

Examples: PL/pgSQL

- Call the function as –

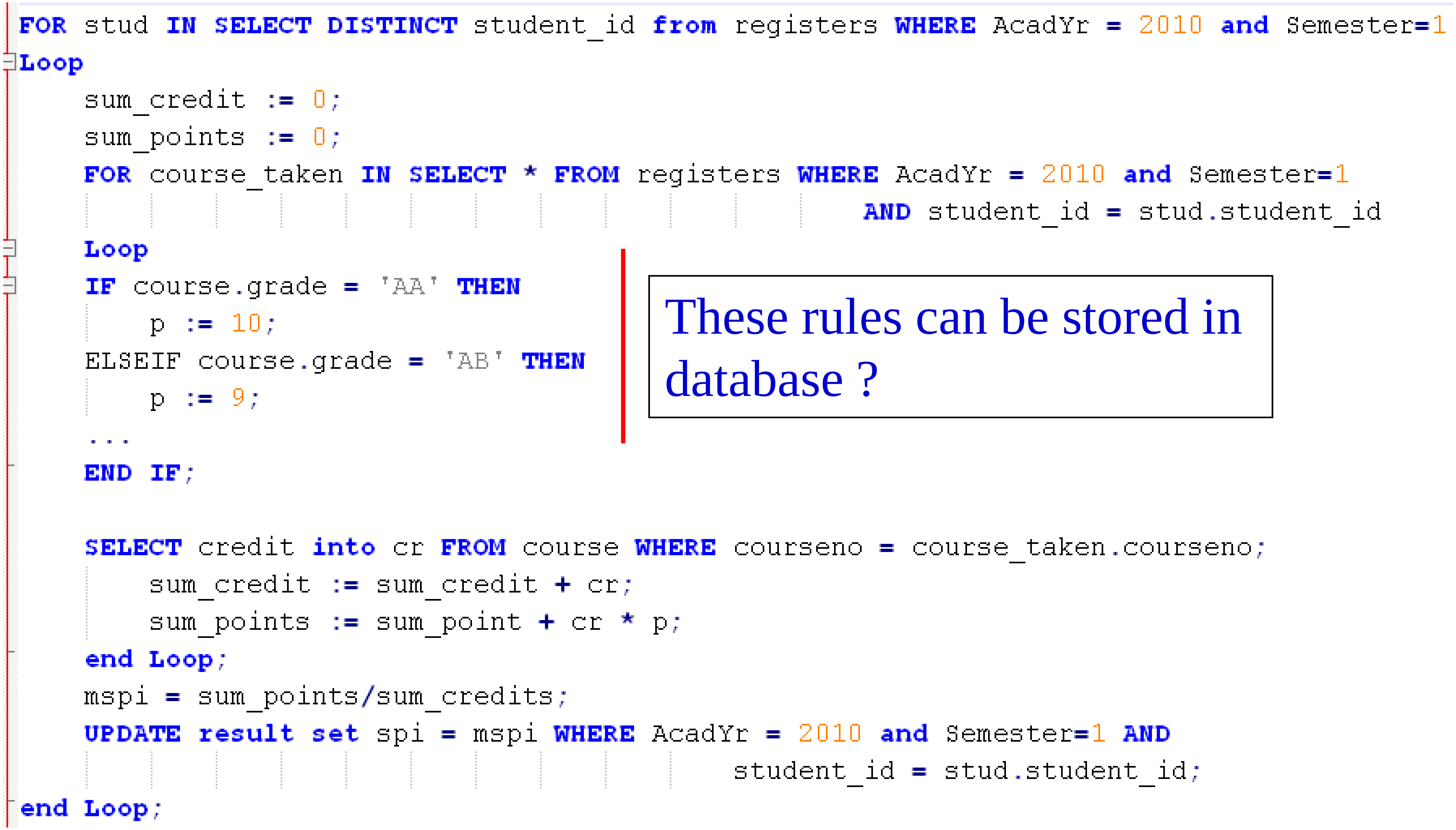
```
SELECT * FROM compute_avg_salary();
```

Partial Code for ComputeSPI



```
FOR stud IN SELECT DISTINCT student_id from registers WHERE AcadYr = 2010 and Semester=1
Loop
    sum_credit := 0;
    sum_points := 0;
    FOR course_taken IN SELECT * FROM registers WHERE AcadYr = 2010 and Semester=1
                                                    AND student_id = stud.student_id
    Loop
    IF course.grade = 'AA' THEN
        p := 10;
    ELSEIF course.grade = 'AB' THEN
        p := 9;
    ...
    END IF;

    SELECT credit into cr FROM course WHERE courseno = course_taken.courseno;
        sum_credit := sum_credit + cr;
        sum_points := sum_point + cr * p;
    end Loop;
    mspi = sum_points/sum_credits;
    UPDATE result set spi = mspi WHERE AcadYr = 2010 and Semester=1 AND
                                            student_id = stud.student_id;
end Loop;
```

Partial Code for ComputeSPI



```
FOR stud IN SELECT DISTINCT student_id from registers WHERE AcadYr = 2010 and Semester=1
Loop
    sum_credit := 0;
    sum_points := 0;
    FOR course_taken IN SELECT * FROM registers WHERE AcadYr = 2010 and Semester=1
                                                 AND student_id = stud.student_id
    Loop
    IF course.grade = 'AA' THEN
        p := 10;
    ELSEIF course.grade = 'AB' THEN
        p := 9;
    ...
    END IF;

    SELECT credit into cr FROM course WHERE courseno = course_taken.courseno;
        sum_credit := sum_credit + cr;
        sum_points := sum_point + cr * p;
    end Loop;
    mspi = sum_points/sum_credits;
    UPDATE result set spi = mspi WHERE AcadYr = 2010 and Semester=1 AND
                                         student_id = stud.student_id;
end Loop;
```

These rules can be stored in database ?

Notion of Cursor

- Cursors are means of iterating through result-set of a query (one by one). It works like a pointer to a query result-set.
- Cursor has two components: Cursor variable and query associated with the cursor variables
- One reason for doing this is to avoid memory overrun when the result contains a large number of rows.
- We have seen a FOR loop iterating through a result-set, it basically uses cursor implicitly, also referred as implicit cursor.

Notion of Cursor

- Explicit cursors definition is a comprehensive mechanism that provide more control over the way result-set is navigated and accessed.

- For example-
  - Can be scrollable non scrollable
  - Updatable or non updatable

Updatable and Scrollable cursor

- If you can update the row being referred by the cursor, then it is “updatable cursor”
- If you go back and forth in a resultset using the cursor then it is scrollable cursor
- In some procedural languages, by default cursors are read-only, and non-scrollable
- PL/PgSQL makes cursor updatable if possible, i.e. for simple (non-join, non-grouping) cursor queries. Default scrollable behavior is also query dependent.

Steps of using cursors (operations of cursors)

- Declare a cursor variable
- Associate (bind) a query with cursor
- Open a cursor
- Fetch a row from the cursor
- Close a cursor

Three ways of creating cursor variables

(1) Unbound

(2) Bound

(3) Parameteric

Bound and Unbound Cursor variables

- Examples:

```
DECLARE
curs1 refcursor; --unbound cursor
curs2 CURSOR FOR
            SELECT * FROM employee;
curs3 CURSOR(pdno integer) IS
            SELECT * FROM employee
                           WHERE dno = pdno;
```

- If you associate a query with cursors at declaration time itself, then it is bound cursor, otherwise it is unbound.
- You associate query with unbound at the time of opening it. You can associate another query with such cursor variables, once done with earlier query.

Opening Unbound Cursors

- OPEN FOR query: if query is static

```
OPEN unbound_cursor [ [ NO ] SCROLL ] FOR query;
```

An example

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

Where curs1 has been declared as following:

```
curs1 refcursor;
```

Opening Unbound Cursors for dynamic queries

- OPEN FOR EXECUTE: when query is created at run-time

```
OPEN unbound_cursor [ [ NO ] SCROLL ] FOR
  EXECUTE query_string;
```

- An example:

```
OPEN curs1 FOR
  EXECUTE 'SELECT * FROM ' || table_name;
```

Opening Bound Cursors

- Opening a Bound Cursor

```
OPEN bound_cursor [(argument_values)];
```

- Examples:

```
OPEN curs2;
OPEN curs3(5); --open cursor for dno=5
```

Where **curs3**, has been declared as

```
curs2 CURSOR FOR
         SELECT * FROM employee;
```

and **curs3**, as

```
curs3 CURSOR(pdno integer) IS
   SELECT * FROM employee WHERE dno = pdno;
```

Declaring Cursor Variables (PL/pgSQL)

- All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type `refcursor`.
- One way to create a cursor variable is just to declare it as a variable of type `refcursor`.
- Another way is to use the cursor declaration syntax, i.e. -

  name [[NO] SCROLL] CURSOR [(arguments)] FOR/IS query;
- SCROLL/NO SCROLL to specify if cursor is scrollable, that means, you can scroll back.
- Cursor can have arguments, which are actually specified while opening it

Fetching a row from cursor

- FETCH

```
FETCH [direction {FROM|IN}] cursor INTO
  target;
```

- Example-

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

Using Cursors

- **MOVE**

```
MOVE [ direction { FROM | IN } ] cursor;
```

- Only the difference with FETCH is, the move, just moves the cursor to new location, we do not capture the row data

- Example-

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
```

Close Cursor

```
CLOSE cursor;
```

- This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.
- An example: `CLOSE curs1;`

Example - Cursor

```
CREATE or replace FUNCTION curs_emp() RETURNS integer AS $$
DECLARE
   c CURSOR (pdno integer) IS SELECT fname, salary FROM
  employee WHERE dno = pdno;
   fname text;
   salary real;
BEGIN
  open c(5);
  LOOP
      FETCH c INTO fname, salary;
      EXIT WHEN NOT FOUND;
      raise notice '% %', fname, salary;
  END LOOP;
  CLOSE c;
  RETURN 1;
END;
```

Example using cursor [Oracle-PL/SQL*]

```
DECLARE
     a T1.e%TYPE;
     b T1.f%TYPE;
     CURSOR T1Cursor IS
               SELECT e, f FROM T1 WHERE e < f;
BEGIN
   OPEN T1Cursor;
   LOOP
        FETCH T1Cursor INTO a, b;
        EXIT WHEN T1Cursor%NOTFOUND;
        DELETE FROM T1 WHERE CURRENT OF T1Cursor;
        INSERT INTO T1 VALUES(b, a);
   END LOOP;
   CLOSE T1Cursor;
END;
```

- *Example from **http://infolab.stanford.edu/~ullman/fcdb/oracle/or-plsql.html**

Example – stored procedure to compute rank

```
DECLARE
   m_rank NUMBER(4) := 1;
   CURSOR s_cur IS
      SELECT * FROM mark
                  ORDER BY marks DESC;
   s_rec mark%ROWTYPE;
BEGIN
   OPEN s_cur;
   LOOP
      FETCH s_cur INTO s_rec;
      EXIT WHEN NOT FOUND s_cur;
      UPDATE mark SET rank = m_rank
            WHERE sid = s_rec.sid;
      m_rank := m_rank + 1;
   END LOOP;
   CLOSE s_cur;
END;
```

You can use updatable cursor. [Postgres supports]*

```
DECLARE
   m_rank NUMBER(4) := 1;
   CURSOR s_cur IS
       SELECT * FROM mark   ORDER BY marks DESC;
   s_rec mark%ROWTYPE;

BEGIN
   OPEN s_cur;
   LOOP
       FETCH s_cur INTO s_rec;
       EXIT WHEN NOT FOUND s_cur;
       UPDATE mark SET rank = m_rank WHERE CURRENT OF s_cur;
       m_rank := m_rank + 1;
   END LOOP;
   CLOSE s_cur;
END;
```

* But, only works for non-join and non-grouping cursor queries

Further reading

- PostgreSQL manual : PL/pgSQL
  http://intranet.daiict.ac.in/~pm_jat/postgres/html/plpgsql.html