

PLpgSQL (i)

- PLpgSQL
- Defining PLpgSQL Functions
- PLpgSQL Examples
- PLpgSQL Gotchas
- Data Types
- Syntax/Control Structures
- SELECT...INTO

❖ PLpgSQL

PLpgSQL = **P**rocedural **L**anguage extensions to **P**ostgre**S**QL

A PostgreSQL-specific language integrating features of:

- procedural programming and SQL programming

Provides a means for **extending DBMS functionality**, e.g.

- implementing constraint checking (triggered functions)
- complex query evaluation (e.g. recursive)
- complex computation of column values
- detailed control of displayed results

Details: PostgreSQL Documentation, Chapter 42

❖ Defining PLpgSQL Functions

PLpgSQL functions are created (and inserted into db) via:

```
CREATE OR REPLACE
    funcName(param1, param2, ....)
    RETURNS rettype
AS $$
DECLARE
    variable declarations
BEGIN
    code for function
END;
$$ LANGUAGE plpgsql;
```

Note: the entire function body is a single SQL string (\$\$...\$\$)

❖ PLpgSQL Examples

Example: function to compute x/y "safely"

```
create or replace function
    div(x integer, y integer) returns integer
as $$
declare
    result integer;      -- variable
begin
    if (y <> 0) then      -- conditional
        result := x/y;    -- assignment
    else
        result := 0;      -- assignment
    end if;
    return result;
end;
$$ language plpgsql;
```

❖ PLpgSQL Examples (cont)

Example: function to compute n!

```
create or replace function
    factorial(n integer) returns integer
as $$
declare
    i integer;
    fac integer := 1;
begin
    for i in 1..n loop
        fac := fac * i;
    end loop;
    return fac;
end;
$$ language plpgsql;
```

❖ PLpgSQL Examples (cont)

Example: function to compute n! recursively

```
create function
    factorial(n integer) returns integer
as $$
begin
    if n < 2 then
        return 1;
    else
        return n * factorial(n-1);
    end if;
end;
$$ language plpgsql;
```

Usage: **select factorial(5);**

❖ PLpgSQL Examples (cont)

Example: handle withdrawal from account and return status message

```
create function
    withdraw(acctNum text, amount integer) returns text
as $$
declare bal integer;
begin
    select balance into bal
    from Accounts
    where acctNo = acctNum;
    if bal < amount then
        return 'Insufficient Funds';
    else
        update Accounts
        set     balance = balance - amount
        where  acctNo = acctNum;
        select balance into bal
        from Accounts
        where  acctNo = acctNum;
        return 'New Balance: ' || bal;
    end if;
end;
$$ language plpgsql;
```


❖ PLpgSQL Gotchas

Some things to beware of:

- doesn't provide any i/o facilities (except **RAISE NOTICE**)
 - the aim is to build computations on tables that SQL alone can't do
- functions are not syntax-checked when loaded into DB
 - you don't find out about the syntax error until "run-time"
- error messages are sometimes not particularly helpful
- functions are defined as strings
 - change of "lexical scope" can sometimes be confusing
- giving params/variables the same names as attributes
 - can avoid by starting all param/var names with underscore

Summary: debugging PLpgSQL can sometimes be tricky.

❖ Data Types

PLpgSQL constants and variables can be defined using:

- standard SQL data types (**CHAR**, **DATE**, **NUMBER**, ...)
- user-defined PostgreSQL data types (e.g. **Point**)
- a special structured record type (**RECORD**)
- table-row types (e.g. **Branches%ROWTYPE** or simply **Branches**)
- types of existing variables (e.g. **Branches.location%TYPE**)

There is also a **CURSOR** type for interacting with SQL.

❖ Data Types (cont)

Variables can also be defined in terms of:

- the type of an existing variable or table column
- the type of an existing table row (implicit **RECORD** type)

Examples:

```
quantity      INTEGER;  
start_qty     quantity%TYPE;  
  
employee      Employees%ROWTYPE;  
-- or  
employee      Employees;  
  
name          Employees.name%TYPE;
```

❖ Syntax/Control Structures

Typical set of control structures, with extensions:

Assignment *var* := *expr*
 SELECT *expr* **INTO** *var*

Selection IF *Cond*₁ THEN *S*₁
 ELSIF *Cond*₂ THEN *S*₂ ...
 ELSE *S* END IF

Iteration LOOP *S* END LOOP
 WHILE *Cond* LOOP *S* END LOOP
 FOR *rec_var* IN *Query* LOOP ...
 FOR *int_var* IN *lo..hi* LOOP ...

*S*_{*i*} = list of PLpgSQL statements, each terminated by semi-colon

❖ SELECT...INTO

Can capture query results via:

```
SELECT  Exp1, Exp2, ..., Expn
INTO    Var1, Var2, ..., Varn
FROM    TableList
WHERE   Condition ...
```

The semantics:

- execute the query as usual
- return "projection list" (*Exp*₁, *Exp*₂, ...) as usual
- assign each *Exp*_{*i*} to corresponding *Var*_{*i*}

❖ SELECT...INTO (cont)

Assigning a simple value via **SELECT . . . INTO**:

```
-- cost is local var, price is attr
select price into cost
from   StockList
where  item = 'Cricket Bat';
cost := cost * (1+tax_rate);
total := total + cost;
```

The current PostgreSQL parser also allows this syntax:

```
select into cost price
from   StockList
where  item = 'Cricket Bat';
```

❖ SELECT...INTO (cont)

Assigning whole rows via **SELECT . . . INTO**:

```
declare
    emp      Employees%ROWTYPE;
    -- alternatively, emp  RECORD;
    eName    text;
    pay      real;
begin
    select * into emp
    from Employees where id = 966543;
    eName := emp.name;
    ...
    select name,salary into eName,pay
    from Employees where id = 966543;
end;
```

❖ SELECT...INTO (cont)

In the case of a PLpgSQL statement like

```
select a into b from R where ...
```

If the selection returns no tuples

- the variable **b** gets the value **NULL**

If the selection returns multiple tuples

- the variable **b** gets the value from the first tuple

❖ SELECT...INTO (cont)

An alternative to check for "no data found"

Use the special variable **FOUND** ...

- local to each function, set false at start of function
- set true if a **SELECT** finds at least one tuple
- set true if **INSERT/DELETE/UPDATE** affects at least one tuple
- otherwise, remains as **FALSE**

Example of use:

```
select a into b from R where ...  
if (not found) then  
    -- handle case where no matching tuples b
```

Produced: 27 Feb 2021