PL/pgSQL Control Structures

An Introduction by a Nooblet

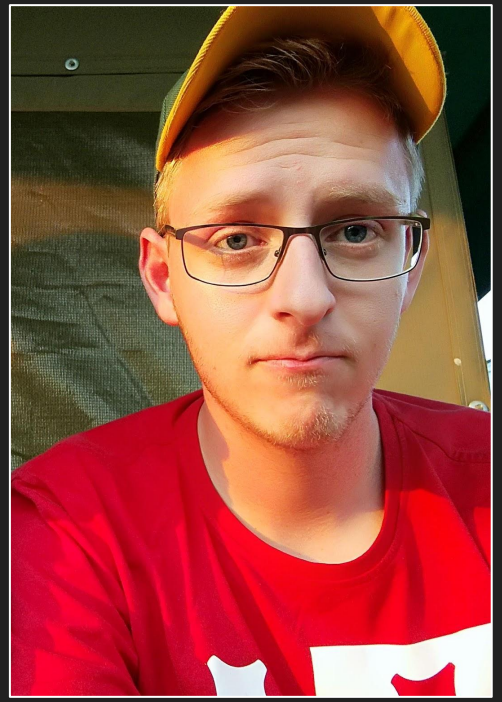
Meet Me,
Andreas Nel

- Software Developer
- Currently employed by Quant Engineering Solutions

https://github.com/AndreasNel

https://www.quora.com/profile/Andreas-Nel-1

https://www.linkedin.com/in/andreas-pp-nel/



What even is the stuff?

PL/pgSQL - A procedural language that can be used to write functions and procedures. Functions encapsulate multiple queries in order to improve performance, and introduces control structures that can be used to perform complex computations.

Control Structure - Block of code that evaluates variables and then performs certain instructions in a specific way based on the given parameters.

Functions vs Procedures, TLDR;

Functions

- Usually have return values
- Is part of a single transaction
- Executed with SELECT

Procedures

- Don’t have return values
- Can start and stop multiple transactions inside
- Executed with CALL

Such Languages

A proper Tower of Babel

- SQL
- PL/pgSQL
- PL/Python
- PL/Java
- PL/R
- PL/PHP
- PL/Ruby
- PL/Scheme
- PL/sh
- PLV8

PLV8

PLV8 is a *trusted* Javascript language extension for PostgreSQL. It can be used for *stored procedures*, *triggers*, etc.

```
CREATE FUNCTION plv8_test(keys TEXT[], vals TEXT[]) RETURNS JSON AS $
    var o = {};
    for(var i=0; i<keys.length; i++){
        o[keys[i]] = vals[i];
    }
    return o;
$ LANGUAGE plv8 IMMUTABLE STRICT;

=# SELECT plv8_test(ARRAY['name', 'age'], ARRAY['Tom', '29']);

 plv8_test
---------------------------
 {"name":"Tom","age":"29"}
(1 row)
```

But Why?

I like overly complicated queries

- Eliminates extra trips between database server and client
- No unnecessary transfer of intermediate results
- Avoids multiple rounds of query parsing

Performance.

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

How to call this function

Positional Notation

- The traditional way
- Arguments are substituted in the order of the parameters
- Arguments can be left out starting from the end

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Named Notation

- Argument names and their values are indicated with => (newer syntax) or := (older syntax)
- Arguments can be specified in any order
- Omitted arguments take their default values
- Cannot be used with aggregate functions, unless it is used as a window function

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Mixed Notation

- Mixture of positional and named notation
- Positional arguments must precede named arguments
- Cannot be used with aggregate functions, unless it is used as a window function

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

Return Statements

Returning to the Mothership

Return statements are used to stop the execution of a function and return to the caller with a result.

A return value can be of a scalar type (int, text, varchar, etc.) or a composite type (row, record) for functions with single return values.

No return value → return VOID.

Functions can also return sets of values (think of it as a function returning a table).

How to Return (single return values)

RETURN *expression*;

- Evaluates expression, terminates function and returns result
- Scalar return values → Cast result to corresponding type
- Composite → Deliver exact specified column set
- Required in functions, except when output parameters are given or the return type is VOID

How to Return (sets of return values)

RETURN NEXT *expression*;

RETURN QUERY *query*;

RETURN QUERY EXECUTE *command-string* [USING *expression* [,…]];

- Only when functions return SETOF *sometype*
- Appends results to result set, exits with final RETURN
- Entire result set kept in memory, (*work_mem* config variable)

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Since execution is not finished, we can check whether rows were returned
    -- and raise exception if not.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END
$BODY$
LANGUAGE plpgsql;

-- Returns available flights or raises exception if there are no
-- available flights.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Conditionals

I have some conditions...

IF … THEN … END IF

IF … THEN … ELSE … END IF

IF … THEN … ELSIF … THEN … ELSE … END IF

CASE … WHEN … THEN … ELSE … END CASE

CASE WHEN … THEN … ELSE … END CASE

- When you want alternative commands to execute based on certain conditions

No boolean short-circuiting

If this then do that

```
IF boolean-expression THEN
    statements
END IF;
```

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

...else do this

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

...elsif elsif elsif elsif elsif elsif

- Conditions are tested successively
- ELSIF == ELSEIF
- Can be accomplished with nested IF-ELSE statements

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

Make your case

Simple CASE

- Search expression evaluated **once**
- Successively compared for equality
- No ELSE and no match → CASE_NOT_FOUND exception

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

...or if you're feeling frisky

Searched CASE

- Each WHEN clause is evaluated in turn
- Only accepts boolean expressions
- Subsequent expressions are not evaluated if found
- No else and no match → CASE_NOT_FOUND exception
- Entirely equivalent to IF-THEN-ELSIF (+ exception)

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

Loops

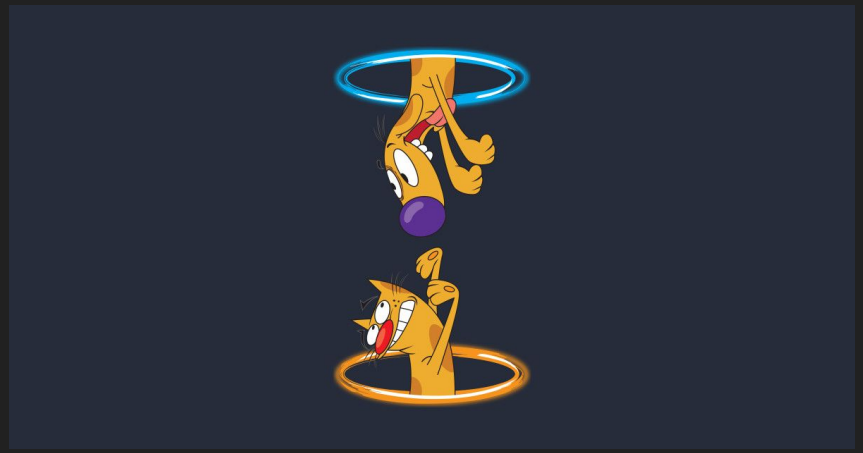
Feeling a little loopy?

```
[ <<label>> ]

LOOP

    statements

END LOOP [ label ];
```



- Unconditional, indefinite looping until a RETURN or EXIT statement executes
- Labels can be used by nested loops to indicate places referred by CONTINUE and EXIT statements

How to interfere

EXIT [label] [WHEN *boolean-expression*];

- Terminates innermost loop
- Unless used with BEGIN block, then a label is mandatory

CONTINUE [label] [WHEN *boolean-expression*];

- Skips following statements
- Starts following iteration of loop

While we’re busy

```
[ <<label>> ]

WHILE boolean-expression LOOP

    statements

END LOOP [ label ];
```

- Executes while expression evaluates to TRUE
- Evaluated before each iteration

It’s for their own good

- Loop variable only exists inside loop, automagically int
- Lower and upper expressions evaluated once
- BY clause indicates step

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;

FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

I love it when you talk FOR-IN to me

```
[ <<label>> ]

FOR target IN query LOOP

    statements

END LOOP [ label ];
```

- Variable declared as row, record, or comma-separated scalar variables, still accessible after loop
- *query* can be any query that returns rows
- PL/pgSQL variables substituted, query plan cached

Execute Order 66

[<<label>>]

FOR *target* IN EXECUTE *text_expression* [USING *expression* [, ...]] LOOP

    statements

END LOOP [label];

- String expression is executed and planned on each iteration
- Smart programmers can adjust speed and/or flexibility of dynamic query

I should get a raise for this

- FOREACH loops through arrays
- Variable can be comma-separated list, each is assigned the corresponding array value
- SLICE indicates the dimension in which array is traversed
- No SLICE → elements traversed in storage order

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

Error Handling

Sometimes, things break

“exceptional circumstances”

Breaking bad?

- Exceptions can be caught and handled
- Local variables accessible
- https://www.postgresql.org/docs/11/errcodes-appendix.html

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

This is not a fix-it-all block!

Tip

A block containing an `EXCEPTION` clause is significantly more expensive to enter and exit than a block without one. Therefore, don't use `EXCEPTION` without need.

What broke?

- SQLSTATE and SQLERRM
- GET STACKED DIAGNOSTICS *variable* { = | := } item [, …];

Name	Type	Description
RETURNED_SQLSTATE	text	the SQLSTATE error code of the exception
COLUMN_NAME	text	the name of the column related to exception
CONSTRAINT_NAME	text	the name of the constraint related to exception
PG_DATATYPE_NAME	text	the name of the data type related to exception
MESSAGE_TEXT	text	the text of the exception's primary message
TABLE_NAME	text	the name of the table related to exception
SCHEMA_NAME	text	the name of the schema related to exception
PG_EXCEPTION_DETAIL	text	the text of the exception's detail message, if any
PG_EXCEPTION_HINT	text	the text of the exception's hint message, if any
PG_EXCEPTION_CONTEXT	text	line(s) of text describing the call stack at the time of the exception (see Section 43.6.9)

Where am I?

Name	Type	Description
ROW_COUNT	bigint	the number of rows processed by the most recent SQL command
RESULT_OID	oid	the OID of the last row inserted by the most recent SQL command (only useful after an INSERT command into a table having OIDs)
PG_CONTEXT	text	line(s) of text describing the current call stack (see Section 43.6.9)

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Call Stack ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

All the Links

- https://www.postgresql.org/docs/11/plpgsql.html
- https://www.postgresql.org/docs/11/errcodes-appendix.html
- https://www.postgresql.org/docs/11/plpgsql-control-structures.html
- All images sampled from Google Images