Copyright

Authors Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk
Translated by Liudmila Mantrova and Alexander Meleshko
Photo by Oleg Bartunov (Tukuche peak, Nepal)

Use of course materials

Feedback
Please send your feedback, comments and suggestions to:
edu@postgrespro.com

Disclaimer

Topics

Using SQL commands in PL/pgSQL code

Eliminating naming ambiguities

Checking command status

Table functions

Returning nothing

SQL commands are embedded into PL/pgSQL code

- same as in expressions:
  the query is prepared, PL/pgSQL variables become parameters

SELECT → PERFORM

- convenient for calling functions with side effects
- WITH queries should be wrapped into a SELECT

INSERT, UPDATE, DELETE and other SQL commands

- except for service commands
- transaction management: only in procedures and anonymous blocks

As we have already seen, PL/pgSQL is very closely integrated with SQL. In particular, all expressions are computed using prepared SQL operators. Besides, expressions can use PL/pgSQL variables and routine parameters: they will be implicitly converted to query parameters.

SQL queries can also be executed within PL/pgSQL code. To execute a query that returns no result (INSERT, UPDATE, DELETE, CREATE, DROP, etc.), you just need to write an SQL command within the PL/pgSQL code as a separate operator.

Commands are prepared exactly like expressions, with PL/pgSQL variables used as parameters. It allows caching the parsed (or planned) query to avoid repeating this work.

In a similar manner, you can also execute a regular SELECT statement if its result is unimportant; simply replace the SELECT keyword with PERFORM. It makes sense for cases like calling functions with side effects. WITH queries have to be wrapped into a SELECT statement (so that it starts with PERFORM when executed).

Service commands like VACUUM, REINDEX and others are not allowed in routines.

COMMIT and ROLLBACK commands are allowed only in procedures and anonymous blocks (executed by the SQL command DO).

https://postgrespro.com/docs/postgresql/17/plpgsql-statements#PLPGSQLSTATEMENTS-GENERAL-SQL

Returning no rows

If the result of the query is not needed, replace SELECT with PERFORM:

```
=> CREATE FUNCTION do_something() RETURNS void
AS $$
BEGIN
    RAISE NOTICE 'Something has been done.';
END
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> DO $$
BEGIN
    PERFORM do_something();
END
$$;
```

```
NOTICE:  Something has been done.
DO
```

Almost any SQL command returning no rows can be used within the PL/pgSQL code without any modifications:

```
=> DO $$
BEGIN
    CREATE TABLE test(n integer);
    INSERT INTO test VALUES (1),(2),(3);
    UPDATE test SET n = n + 1 WHERE n > 1;
    DELETE FROM test WHERE n = 1;
    DROP TABLE test;
END
$$;
```

```
DO
```

Transaction control in procedures

Procedures (and anonymous blocks) written in PL/pgSQL support transaction control commands:

```
=> CREATE TABLE test(n integer);
```

```
CREATE TABLE
```

```
=> CREATE PROCEDURE foo()
AS $$
BEGIN
    INSERT INTO test VALUES (1);
    COMMIT;
    INSERT INTO test VALUES (2);
    ROLLBACK;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CALL foo();
```

```
CALL
```

```
=> SELECT * FROM test;
```

```
 n
---
 1
(1 row)
```

There are certain limitations. First of all, in such cases, a procedure itself must start a new transaction, i.e., it must not be executed in the context of an already started transaction.

```
=> BEGIN;
```

```
BEGIN
```

```
=> CALL foo(); -- error
```

```
ERROR:  invalid transaction termination
CONTEXT:  PL/pgSQL function foo() line 4 at COMMIT
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Second, the call stack of this procedure must contain nothing but CALL operators.

In other words, if a procedure calls a procedure... that calls a procedure that performs transaction control, everything works fine:

```
=> CREATE OR REPLACE PROCEDURE foo()
AS $$
BEGIN
    CALL bar();
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CREATE PROCEDURE bar()
AS $$
BEGIN
    CALL baz();
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CREATE PROCEDURE baz()
AS $$
BEGIN
    COMMIT;
END
$$ LANGUAGE plpgsql;
```

```
CREATE PROCEDURE
```

```
=> CALL foo(); -- it works
```

```
CALL
```

But should this stack include, say, a function call, the transaction would have to be completed somewhere in the middle of the operator which the context belongs to, i.e., SELECT. This is prohibited, so we get the error:

```
=> CREATE FUNCTION qux() RETURNS void
AS $$
BEGIN
    CALL bar();
END
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT qux(); -- error
```

```
ERROR:  invalid transaction termination
CONTEXT:  PL/pgSQL function baz() line 3 at COMMIT
SQL statement "CALL baz()"
PL/pgSQL function bar() line 3 at CALL
SQL statement "CALL bar()"
PL/pgSQL function qux() line 3 at CALL
```

Returning a single row

SELECT … INTO

- get the first returned row
- one variable of a composite type
  or an appropriate number of scalar variables

INSERT, UPDATE, DELETE RETURNING … INTO

- get the inserted (updated, deleted) row
- one variable of a composite type
  or an appropriate number of scalar variables

If the query result is important, you can use the INTO clause to save it into a single variable of a composite type or into several scalar variables. If the query returns several rows, only the first one makes it into the variable (you can set the output order with the ORDER BY clause). If the query returns no rows, the variable will be set to NULL.

In a similar manner, you can also use INSERT, UPDATE, and DELETE commands with the RETURNING clause. The difference is that these commands must not return more than one row: it will lead to an error as there is no way to specify which row is the “first”.

https://postgrespro.com/docs/postgresql/17/plpgsql-statements#PLPGSQL-STATEMENTS-SQL-ONEROW

If a query returns several rows, but only one of them is used, it’s highly likely that this query is incorrect. PL/pgSQL can report such suspicious situations (and several others).

https://postgrespro.com/docs/postgresql/17/plpgsql-development-tips#PLPGSQL-EXTRA-CHECKS

More powerful debugging capabilities are provided by an external extension plpgsql_check (developed by Pavel Stehule).

https://github.com/okbob/plpgsql_check

Returning a single row

A SELECT command returning a single row is probably the most frequently used one in PL/pgSQL. Here is an example that returns one row, despite the query within the anonymous block returning two:

```
=> CREATE TABLE t(id integer, code text);
```

```
CREATE TABLE
```

```
=> INSERT INTO t VALUES (1, 'One'), (2, 'Two');
```

```
INSERT 0 2
```

```
=> DO $$
DECLARE
    r record;
BEGIN
    SELECT id, code INTO r FROM t;
    RAISE NOTICE '%', r;
END
$$;
```

```
NOTICE:  (1,One)
DO
```

Note the construct in the PostgreSQL code that is very similar to SELECT INTO from SQL. Here, after INTO, there is the name of the new table, created and filled with the query output. In PL/pgSQL, the equivalent syntax would be CREATE TABLE ... AS SELECT.

INSERT, UPDATE, and DELETE commands can also return the result using the RETURNING clause. They can be used in PL/pgSQL with the INTO clause, just like SELECT:

```
=> DO $$
DECLARE
    r record;
BEGIN
    UPDATE t SET code = code || '!' WHERE id = 1 RETURNING * INTO r;
    RAISE NOTICE 'Modified: %', r;
END
$$;
```

```
NOTICE:  Modified: (1,One!)
DO
```

Compile-time and run-time checks for routines

In some suspicious cases, PL/pgSQL can issue warnings. Warnings can be enabled with the following parameter (set to none by default):

```
=> SET plpgsql.extra_warnings = 'all';
```

```
SET
```

```
=> CREATE PROCEDURE bugs(INOUT a integer)
AS $$
DECLARE
    a integer;
    b integer;
BEGIN
    SELECT id INTO a, b FROM t;
END
$$ LANGUAGE plpgsql;
```

```
WARNING:  variable "a" shadows a previously defined variable
LINE 4:     a integer;
            ^
CREATE PROCEDURE
```

This is a warning about two variable declarations that override each other:

```
=> CALL bugs(42);
```

```
WARNING:  query returned more than one row
HINT:  Make sure the query returns a single row, or use LIMIT 1.
WARNING:  number of source and target fields in assignment does not match
DETAIL:  strict_multi_assignment check of extra_warnings is active.
HINT:  Make sure the query returns the exact list of columns.
 a
----
 42
(1 row)
```

And here we can see two run-time warnings: the query has returned more than one row, and the INTO clause provides the wrong number of parameters (PL/pgSQL will assign NULL to the second one). There are no other checks available at the moment, but they can appear in the next PostgreSQL versions.

The plpgsql.extra_warnings value can specify some particular checks. A similar parameter plpgsql.extra_errors enables throwing errors instead of warnings.

```
=> RESET plpgsql.extra_warnings;
```

```
RESET
```

You can perform more extensive code checks using plpgsql_check, an external extension developed and supported by Pavel Stehule. This extension is already installed in the VM provided for this course.

```
=> CREATE SCHEMA plpgsql_check;
```

```
CREATE SCHEMA
```

```
=> CREATE EXTENSION plpgsql_check SCHEMA plpgsql_check;
```

```
CREATE EXTENSION
```

```
=> SELECT * FROM plpgsql_check.plpgsql_check_function('bugs(integer)');
```

```
                       plpgsql_check_function
-----------------------------------------------------------------------
 warning:00000:5:statement block:parameter "a" is shadowed
 Detail: Local variable shadows function parameter.
 warning:00000:6:SQL statement:too few attributes for target variables
 Detail: There are more target variables than output columns in query.
 Hint: Check target variables in SELECT INTO statement.
 warning extra:00000:3:DECLARE:never read variable "a"
 warning extra:00000:4:DECLARE:never read variable "b"
 warning extra:00000:unused parameter "a"
 warning extra:00000:unmodified OUT variable "a"
(9 rows)
```

Unused variables and an output parameter without an assigned value have been detected.

The plpgsql_checks extension provides many debugging features, including run-time error detection. It also includes a profiler for tuning PL/pgSQL code.

Eliminating naming ambiguities

Will the following code be executed successfully?

```
=> DO $$
DECLARE
    id   integer := 1;
    code text;
BEGIN
    SELECT id, code INTO id, code
    FROM t WHERE id = id;
    RAISE NOTICE '%, %', id, code;
END
$$;
```

```
ERROR:  column reference "id" is ambiguous
LINE 1: SELECT id, code                  FROM t WHERE id = id
               ^
DETAIL:  It could refer to either a PL/pgSQL variable or a table column.
QUERY:  SELECT id, code                  FROM t WHERE id = id
CONTEXT:  PL/pgSQL function inline_code_block line 6 at SQL statement
```

No, because of the ambiguity in SELECT: id can mean both the name of the column and the name of the variable. Note that there is no ambiguity in the INTO clause: it relates to PL/pgSQL only. By the way, the message shows that PL/pgSQL cuts off the INTO clause before passing the query to SQL.

There are several approaches to eliminating such ambiguities.

The first one is to prevent them. It can be achieved by prepending variable names with a prefix, which usually corresponds to the variable type, for example:

- p_ for parameters
- l_ or v_ for regular (local) variables
- c_ for constants

It is a simple and effective method if you use it consistently and never add prefixes to column names. Its disadvantage is that it makes the code look sloppy and somewhat lively because of extra underscores.

This is how it can look like in our example:

```
=> DO $$
DECLARE
    l_id   integer := 1;
    l_code text;
BEGIN
    SELECT id, code INTO l_id, l_code
    FROM t WHERE id = l_id;
    RAISE NOTICE '%, %', l_id, l_code;
END
$$;
```

```
NOTICE:  1, One!
DO
```

Another approach is to use qualified names, i.e., prepend the object name with a qualifier followed by a dot:

- a name or an alias of a table for columns,
- a block label for variables,
- a function name for parameters.

This approach is more straightforward than adding prefixes because it works for any column names.

This is how our example will look like if we use qualifiers:

```
=> DO $$
<<local>>
DECLARE
    id   integer := 1;
    code text;
BEGIN
    SELECT t.id, t.code INTO local.id, local.code
    FROM t WHERE t.id = local.id;
    RAISE NOTICE '%, %', id, code;
END
$$;
```

```
NOTICE:  1, One!
DO
```

The third approach is to prioritize variables over columns, or vice versa. Such prioritization is managed by the plpgsql.variable_conflict configuration parameter. Its possible values are use_column, use_variable, error.

In some cases, it facilitates conflict resolution, but it does not eliminate conflicts altogether. Besides, such an implicit rule (which can suddenly change, to make things worse) is sure to cause some code to be executed in a way unforeseen by the developer.

Nevertheless, let’s see an example. Here variables have priority, so it is enough to add qualifiers to table columns only:

```
=> SET plpgsql.variable_conflict = use_variable;
```

```
SET
```

```
=> DO $$
DECLARE
    id   integer := 1;
    code text;
BEGIN
    SELECT t.id, t.code INTO id, code
    FROM t WHERE t.id = id;
    RAISE NOTICE '%, %', id, code;
END
$$;
```

```
NOTICE:  1, One!
DO
```

```
=> RESET plpgsql.variable_conflict;  -- reset the value to default
```

```
RESET
```

```
=> SHOW plpgsql.variable_conflict;
```

```
 plpgsql.variable_conflict
---------------------------
 error
(1 row)
```

Which approach to follow is a matter of taste and experience. We recommend choosing either the first or the second one (prefixes or qualifiers); make sure not to use both in one project because consistency can greatly assist code comprehension.

In this course, we are going to use qualifiers, but only where it's absolutely necessary, so that examples don't get too cluttered.

However, in production code you should always take care of all possible ambiguities: there is no guarantee that a new column won't have the same name as your variable.

Checking results

INTO STRICT

guarantees that exactly one row is returned, no more and no less

ROW_COUNT diagnostics

the number of rows returned (processed) by the last SQL command

FOUND variable

after an SQL command: true if the command has returned (processed) a row

after a loop: indicates that at least one iteration has been completed

By adding the STRICT keyword to the INTO clause, we can guarantee that the command returns or processes exactly one row; otherwise, an error occurs.

Besides, we can check the status of the SQL command that has just been executed (unless it has completed with an error). There are two ways to go about it.

The first option is to add the GET DIAGNOSTICS clause that returns the number of rows processed by this command.

The second option is to use the boolean variable FOUND, which shows whether the command has processed any data at all.

You can also use FOUND as an indicator that the loop body has been executed at least once.

https://postgrespro.com/docs/postgresql/17/plpgsql-statements#PLPGSQL-STATEMENTS-DIAGNOSTICS

Exactly one row

What will happen if the query returns several rows?

```
=> DO $$
DECLARE
    r record;
BEGIN
    SELECT id, code INTO r FROM t;
    RAISE NOTICE '%', r;
END
$$;
```

```
NOTICE:  (2,Two)
DO
```

Only the first row will be written into the variable. Since we have not specified the ORDER BY clause, the order is virtually unpredictable:

```
=> SELECT * FROM t;
```

```
 id | code
----+------
  2 | Two
  1 | One!
(2 rows)
```

INSERT, UPDATE, and DELETE commands do not allow specifying the order of the rows, so a command that affects several rows results in an error:

```
=> DO $$
DECLARE
    r record;
BEGIN
    UPDATE t SET code = code || '!' RETURNING * INTO r;
    RAISE NOTICE 'Modified: %', r;
END
$$;
```

```
ERROR:  query returned more than one row
HINT:  Make sure the query returns a single row, or use LIMIT 1.
CONTEXT:  PL/pgSQL function inline_code_block line 5 at SQL statement
```

And what if the query returns no rows at all?

```
=> DO $$
DECLARE
    r record;
BEGIN
    r := (-1,'!!!');
    SELECT id, code INTO r FROM t WHERE false;
    RAISE NOTICE '%', r;
END
$$;
```

```
NOTICE:  (,)
DO
```

Variables will contain undefined values.

It is also true for INSERT, UPDATE, and DELETE commands. For example:

```
=> DO $$
DECLARE
    r record;
BEGIN
    UPDATE t SET code = code || '!' WHERE id = -1
        RETURNING * INTO r;
    RAISE NOTICE 'Modified: %', r;
END
$$;
```

```
NOTICE:  Modified: (,)
DO
```

Sometimes you may want to be sure that the query retrieves exactly one row, no more and no less. In this case, it is convenient to use the INTO STRICT clause:

```
=> DO $$
DECLARE
    r record;
BEGIN
    SELECT id, code INTO STRICT r FROM t;
    RAISE NOTICE '%', r;
END
$$;
```

```
ERROR:  query returned more than one row
HINT:  Make sure the query returns a single row, or use LIMIT 1.
CONTEXT:  PL/pgSQL function inline_code_block line 5 at SQL statement
```

```
=> DO $$
DECLARE
    r record;
BEGIN
    SELECT id, code INTO STRICT r FROM t WHERE false;
    RAISE NOTICE '%', r;
END
$$;
```

```
ERROR:  query returned no rows
CONTEXT:  PL/pgSQL function inline_code_block line 5 at SQL statement
```

As we have seen, INSERT, UPDATE, and DELETE commands that affect several rows result in an error. The STRICT keyword guarantees that there will be exactly one row (and not zero):

```
=> DO $$
DECLARE
    r record;
BEGIN
    UPDATE t SET code = code || '!' WHERE id = -1 RETURNING * INTO STRICT r;
    RAISE NOTICE 'Modified: %', r;
END
$$;
```

```
ERROR:  query returned no rows
CONTEXT:  PL/pgSQL function inline_code_block line 5 at SQL statement
```

Explicit checks of the execution state

You can also check the state of the last SQL command executed:

- The GET DIAGNOSTICS command retrieves the number of processed rows (ROW_COUNT).
- A predefined boolean variable FOUND shows whether any row has been processed.

```
=> DO $$
DECLARE
    r record;
    rowcount integer;
BEGIN
    SELECT id, code INTO r FROM t WHERE false;

    GET DIAGNOSTICS rowcount := ROW_COUNT;
    RAISE NOTICE 'rowcount = %', rowcount;
    RAISE NOTICE 'found = %', FOUND;
END
$$;
```

```
NOTICE:  rowcount = 0
NOTICE:  found = f
DO
```

```
=> DO $$
DECLARE
    r record;
    rowcount integer;
BEGIN
    SELECT id, code INTO r FROM t;

    GET DIAGNOSTICS rowcount := ROW_COUNT;
    RAISE NOTICE 'rowcount = %', rowcount;
    RAISE NOTICE 'found = %', FOUND;
END
$$;
```

```
NOTICE:  rowcount = 1
NOTICE:  found = t
DO
```

Note: such diagnostics does not detect that the query affects several rows. ROW_COUNT returns 1, since only one row from the output was saved to the variable r.

Set returning functions

Query rows

```
RETURN QUERY query;
```

One row

`RETURN NEXT expression;`	*if there are no output parameters*
`RETURN NEXT;`	*if there are output parameters*

Features

rows are added to the result, but function execution is not terminated
commands can be executed several times
the result is not returned until the function completes

9

To create a table function in PL/pgSQL, you have to declare it with a clause RETURNS SETOF or RETURNS TABLE (just like in SQL).

Commands RETURN QUERY and RETURN NEXT assemble individual elements of the returning value in a special buffer. These commands do not interrupt execution. They can be called multiple times within a function body, each call adding elements to the final output. When a function finishes (by a RETURN call with no parameters or simply by reaching the end of the function body), the output that has been assembled in the buffer is returned.

If a function finishes after executing the RETURN QUERY *query* command once, the output will be identical to that of an SQL function running the *query* as its last. However, the query from an SQL function could be substituted into the main query, while with PL/pgSQL this is not the case.

The result can also be returned row by row using the RETURN NEXT construct. It is similar to an ordinary RETURN, but instead of stopping the function execution, it adds the return value as another row of the future result.

Thus, the RETURN NEXT and RETURN QUERY commands do *not* work as `yield` in the generator functions of modern languages.

https://postgrespro.com/docs/postgresql/17/plpgsql-control-structures#PLPGSQL-STATEMENTS-RETURNING

Set returning functions

Here is an example of a set returning function written in PL/pgSQL:

```
=> CREATE FUNCTION t() RETURNS TABLE(LIKE t)
AS $$
BEGIN
    RETURN QUERY SELECT id, code FROM t ORDER BY id;
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT * FROM t();
```

```
 id | code
----+------
  1 | One!
  2 | Two
(2 rows)
```

Another option is to return values row by row.

```
=> CREATE FUNCTION days_of_week() RETURNS SETOF text
AS $$
BEGIN
    FOR i IN 7 .. 13 LOOP
        RETURN NEXT to_char(to_date(i::text,'J'),'TMDy');
    END LOOP;
END;
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> SELECT * FROM days_of_week() WITH ORDINALITY;
```

```
 days_of_week | ordinality
--------------+-----------
 Mon          |          1
 Tue          |          2
 Wed          |          3
 Thu          |          4
 Fri          |          5
 Sat          |          6
 Sun          |          7
(7 rows)
```

Why is this function declared STABLE, not IMMUTABLE?

At first sight, it seems that repeat function calls will return the same value, it is still affected by the current locale:

```
=> SET lc_time = 'ne_NP.UTF8';
```

```
SET
```

```
=> SELECT * FROM days_of_week() WITH ORDINALITY;
```

```
 days_of_week | ordinality
--------------+-----------
 सोम          |          1
 मङ्गल         |          2
 बुध           |          3
 बिही         |          4
 शुक्र          |          5
 शनि          |          6
 आइत           |          7
(7 rows)
```

The following example shows a mixed approach: create a function that returns a list of routines that belong to its schema. It uses PG_ROUTINE_OID to get its oid:

```
=> CREATE FUNCTION where_am_i() RETURNS TABLE(name text, isitme text)
AS $$
DECLARE

    my_oid oid;
    schema_oid oid;
BEGIN
    GET DIAGNOSTICS my_oid := PG_ROUTINE_OID;
    -- schema oid
    schema_oid := pronamespace FROM pg_proc WHERE oid = my_oid;
    -- Header
    name := '=== Schema: ' || schema_oid::regnamespace;
    RETURN NEXT;  -- header with shcema name
    -- list of routines
    RETURN QUERY
                SELECT proname::text, CASE WHEN oid = my_oid THEN 'It''s me!' END
                FROM pg_proc
                WHERE pronamespace = schema_oid
                ORDER BY 1;
END
$$ STABLE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Run the function:

```
=> SELECT * FROM where_am_i();
```

```
        name        |  isitme
--------------------+----------
 === Schema: public |
 bar                |
 baz                |
 bugs               |
 days_of_week       |
 do_something       |
 foo                |
 qux                |
 t                  |
 where_am_i         | It's me!
(10 rows)
```

Takeaways

PL/pgSQL is closely integrated with SQL

- procedural code can contain queries (provided as expressions or as separate commands)
- queries can include variables
- you can get query results and check query status

Naming ambiguities are an issue

Practice

1. Create the add_author function for adding new authors. The function must take three parameters (last name, first name, middle name) and return the ID of the added author.

   Check that the application allows adding new authors.

2. Create the buy_book function for buying books. The function takes the book ID as a parameter and reduces the number of such books by one. There is no return value.

   Check that the Store tab now allows buying books.

12

1.

```
FUNCTION add_author(last_name text, first_name text, middle_name
text)
RETURNS integer
```

3.

```
FUNCTION buy_book(book_id integer)
RETURNS void
```

You can notice that the application allows selling more books than there are in store. If the total number of books were stored in a table column, adding a CHECK constraint would be a good and simple solution. But our implementation calculates the number of books, so we are not going to address this until we get to the “Triggers” section of the course.

Task 1. The add_author function

```
=> CREATE FUNCTION add_author(
    last_name text,
    first_name text,
    middle_name text
) RETURNS integer
AS $$
DECLARE
    author_id integer;
BEGIN
    INSERT INTO authors(last_name, first_name, middle_name)
        VALUES (last_name, first_name, middle_name)
        RETURNING authors.author_id INTO author_id;
    RETURN author_id;
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

Task 2. The buy_book function

```
=> CREATE FUNCTION buy_book(book_id integer)
RETURNS void
AS $$
BEGIN
    INSERT INTO operations(book_id, qty_change)
        VALUES (book_id, -1);
END
$$ VOLATILE LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```