Авторские права
© Postgres Professional, 2017–2020
Авторы: Егор Рогов, Павел Лузанов

Использование материалов курса
Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Отказ от ответственности
Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или непрямым, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Темы

Использование команд SQL в коде PL/pgSQL

Устранение неоднозначностей именования

Проверка статуса команды

Табличные функции

Команды без результата

Команды SQL встраиваются в код PL/pgSQL

- как и в выражениях:
  запрос подготавливается,
  переменные PL/pgSQL подставляются как параметры

SELECT → PERFORM

- удобно для вызова функций с побочными эффектами
- запросы, начинающиеся на WITH, надо «оборачивать» в SELECT

INSERT, UPDATE, DELETE и другие команды SQL

- кроме служебных команд
- управление транзакциями — только в процедурах и анонимных блоках

Как мы уже видели, PL/pgSQL очень тесно интегрирован с SQL. В частности, все выражения вычисляются с помощью подготовленных SQL-операторов к базе данных. При этом в выражениях можно использовать переменные PL/pgSQL и параметры подпрограмм — они автоматически подставляются в запрос в виде параметров.

Внутри кода на PL/pgSQL можно выполнять и SQL-запросы. Чтобы выполнить запрос, не возвращающий результат (INSERT, UPDATE, DELETE, CREATE, DROP и т. п.), достаточно просто написать команду SQL внутри кода на PL/pgSQL как отдельный оператор.

Команды подготавливаются точно так же, как и выражения. Это позволяет закешировать разобранный (или спланированный) запрос и не выполнять эту часть работы повторно. Внутри команд так же можно использовать переменные PL/pgSQL, которые будут автоматически заменены на параметры.

Таким же образом можно вызвать и обычный запрос SELECT, если его результат не важен — для этого ключевое слово SELECT надо заменить на PERFORM. Это имеет смысл, например, для вызова функций с побочным эффектом. Если запрос начинается с WITH, его необходимо «обернуть» в SELECT (чтобы в конечном итоге запрос начинался на PERFORM).

Команды COMMIT и ROLLBACK допускаются только в процедурах и в анонимных блоках кода (выполняемых SQL-командой DO).

https://postgrespro.ru/docs/postgresql/12/plpgsql-statements#PLPGSQL-STATEMENTS-SQL-NORESULT

Команды, не возвращающие результат

Если результат запроса не нужен, заменяем SELECT на PERFORM:

```
=> CREATE FUNCTION do_something() RETURNS void
AS $$
BEGIN
    RAISE NOTICE 'Что-то сделалось.';
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

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

```
NOTICE:  Что-то сделалось.
DO
```

Внутри PL/pgSQL можно использовать без изменений практически любые команды SQL, не возвращающие результат:

```
=> 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
```

Управление транзакциями в процедурах

В процедурах (и в анонимных блоках кода) на PL/pgSQL можно также использовать команды управления транзакциями:

```
=> 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)
```

Действуют определенные ограничения. Во-первых, процедура должна начинать новую транзакцию, то есть не должна выполняться в контексте уже начатой ранее.

```
=> BEGIN;
```

```
BEGIN
```

```
=> CALL foo(); -- ошибка

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

=> ROLLBACK;

ROLLBACK
```

Во-вторых, в стеке вызовов процедуры не должно быть ничего, кроме операторов CALL.

Иными словами, если процедура вызывает процедуру, которая вызывает процедуру... которая выполняет команду управления транзакцией, то все работает:

```
=> 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(); -- работает

CALL
```

Но стоит в этой цепочке появиться, например, вызову функции, как получается, что транзакция должна завершиться где-то «посередине» оператора SELECT, а это недопустимо:

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

CREATE FUNCTION

=> SELECT qux(); -- ошибка

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
```

Одна строка результата

SELECT … INTO

получение первой по порядку строки результата

одна переменная составного типа
или подходящее количество скалярных переменных

INSERT, UPDATE, DELETE RETURNING … INTO

получение вставленной (измененной, удаленной) строки

одна переменная составного типа
или подходящее количество скалярных переменных

Если результат запроса важен, его можно получить с помощью фразы INTO в одну переменную составного типа или в несколько скалярных переменных. Если запрос возвращает несколько строк, в переменную попадет только первая из них (порядок можно гарантировать фразой ORDER BY). Если запрос не возвращает ни одной строки, переменная получит неопределенное значение.

Аналогично можно использовать команды INSERT, UPDATE, DELETE с фразой RETURNING. Отличие состоит в том, что эти команды не должны возвращать более одной строки — это приведет к ошибке, так как нет способа указать, какая из строк считается «первой».

https://postgrespro.ru/docs/postgresql/12/plpgsql-statements#PLPGSQL-STATEMENTS-SQL-ONEROW

Если запрос возвращает несколько строк, из которых используется только одна, это скорее всего говорит о том, что запрос написан неправильно. PL/pgSQL может сообщать о такой подозрительной ситуации (и о нескольких других).

https://postgrespro.ru/docs/postgresql/12/plpgsql-development-tips#PLPGSQL-EXTRA-CHECKS

Более широкими возможностями обладает стороннее расширение plpgsql_check (автор — Павел Стехуле).

https://github.com/okbob/plpgsql_check

Команды, возвращающие одну строку

Наверное, наиболее часто используется в PL/pgSQL команда SELECT, возвращающая одну строку. Пример, который не получилось бы выполнить с помощью выражения с подзапросом (потому что возвращаются сразу две строки):

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

```
CREATE TABLE
```

```
=> INSERT INTO t VALUES (1, 'Раз'), (2, 'Два');
```

```
INSERT 0 2
```

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

```
NOTICE:  (1,Раз)
DO
```

Команды INSERT, UPDATE, DELETE тоже могут возвращать результат с помощью фразы RETURNING. Их можно использовать в PL/pgSQL точно так же, как SELECT, добавив фразу INTO:

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

```
NOTICE:  Изменили: (1,Раз!)
DO
```

Проверки при создании и при выполнении подпрограмм

PL/pgSQL может выдавать предупреждения в некоторых подозрительных случаях. Для этого надо установить параметр (значение по умолчанию — none):

```
=> 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
```

Это предупреждение о перекрывающих друг друга определениях переменных.

```
=> 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)
```

А здесь мы видим два предупреждения времени выполнения: запрос вернул более одной строки, а в предложении INTO указано неверное число параметров (PL/pgSQL присвоит второму неопределенное значение). Других проверок в настоящее время не предусмотрено, но они могут появиться в следующих версиях PostgreSQL.

Значение параметра plpgsql.extra_warnings можно ограничить только определенными проверками. Аналогичный параметр plpgsql.extra_errors будет приводить не к предупреждениям, а к ошибкам.

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

```
RESET
```

Стороннее расширение plpgsql_check, написанное и развиваемое Павлом Стехуле, позволяет проверить код более детально. Расширение уже установлено в виртуальной машине курса.

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

```
CREATE EXTENSION
```

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

```
                       plpgsql_check_function
------------------------------------------------------------------------
 warning:00000:5:statement block:parameter "a" is overlapped
 Detail: Local variable overlap 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)
```

Дополнительно обнаружены неиспользуемые переменные и тот факт, что выходному параметру не присваивается значение.

Расширение имеет множество возможностей для обнаружения проблем в коде, в том числе и на этапе выполнения. Кроме того, расширение включает и профилировщик для целей оптимизации PL/pgSQL-кода.

Устранение неоднозначностей именования

Получится ли выполнить следующий код?

```
=> 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
```

Не получится из-за неоднозначности в SELECT: id может означать и имя столбца, и имя переменной:

Причем во фразе INTO неоднозначности нет — она относится только к PL/pgSQL. В сообщении, кстати, видно, как PL/pgSQL вырезает фразу INTO, прежде чем передать запрос в SQL.

Есть несколько подходов к устранению неоднозначностей.

Первый состоит в том, чтобы неоднозначностей не допускать. Для этого к переменным добавляют префикс, который

обычно выбирается в зависимости от «класса» переменной, например:

* Для параметров p_ (parameter);

* Для обычных переменных l_ (local) или v_ (variable);

* Для констант c_ (constant);

Это простой и действенный способ, если использовать его систематически и никогда не использовать префиксы в именах столбцов. К минусам можно отнести некоторую неряшливость и пестроту кода из-за лишних подчеркиваний.

Вот как это может выглядеть в нашем случае:

```
=> 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, Раз!
DO
```

Второй способ состоит в использовании квалифицированных имен — к имени объекта через точку дописывается уточняющий квалификатор:

* Для столбца — имя или псевдоним таблицы;

* Для переменной — метку блока;

* Для параметра — имя функции.

Такой способ более «честный», чем добавление префиксов, поскольку работает для любых названий столбцов.

Вот как будет выглядеть наш пример с использованием квалификаторов:

```
=> 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, Раз!
DO
```

Третий вариант — установить приоритет переменных над столбцами или наоборот, столбцов над переменными. За это отвечает конфигурационный параметр plpgsql.variable_conflict.

В ряде случаев это упрощает разрешение конфликтов, но не устраняет их полностью. Кроме того, неявное правило (которое, к тому же, может внезапно поменяться) непременно приведет к тому, что какой-то код будет выполняться не так, как предполагал разработчик.

Тем не менее приведем пример. Здесь устанавливается приоритет переменных, поэтому достаточно квалифицировать только столбцы таблицы:

```
=> 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, Раз!
DO
```

```
=> RESET plpgsql.variable_conflict;
```

```
RESET
```

Какой способ выбрать — дело опыта и вкуса. Мы рекомендуем остановиться либо на первом (префиксы), либо на втором (квалификаторы), и не смешивать их в одном проекте, поскольку систематичность крайне важна для облегчения понимания кода.

В курсе мы будем использовать второй способ, но только в тех случаях, когда это действительно необходимо — чтобы не загромождать примеры.

Однако в коде, предназначенном для промышленной эксплуатации, думать о неоднозначностях надо всегда: нет никакой гарантии, что завтра в таблице не появится новый столбец с именем, совпадающим с вашей переменной!

Проверка результата

INTO STRICT

гарантия получения ровно одной строки — ни больше, ни меньше

Диагностика ROW_COUNT

число строк, возвращенных (обработанных) последней командой SQL

Переменная FOUND

после команды SQL: истина, если команда вернула (обработала) строку

после цикла: признак того, что выполнилась хотя бы одна итерация

Добавив к предложению INTO ключевое слово STRICT, мы получим гарантирую того, что команда вернула или обработала *ровно* одну строку — иначе будет зафиксирована ошибка.

Кроме того, можно узнать статус только что выполненной команды SQL (если она не завершилась ошибкой). Для этого есть два способа.

Во-первых, можно получить число строк, затронутых командой, с помощью конструкции GET DIAGNOSTICS.

Во-вторых, специальная логическая переменная FOUND показывает, были ли обработаны командой хотя бы какие-то данные.

Переменную FOUND можно использовать и как индикатор того, что тело цикла выполнилось минимум один раз.

https://postgrespro.ru/docs/postgresql/12/plpgsql-statements#PLPGSQL-STATEMENTS-DIAGNOSTICS

Ровно одна строка

Что произойдет, если запрос вернет несколько строк?

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

```
NOTICE:  (2,Два)
DO
```

В переменную будет записана только первая строка. Поскольку мы не указали ORDER BY, то порядок строк в общем случае непредсказуем:

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

```
 id | code
----+------
  2 | Два
  1 | Раз!
(2 rows)
```

Поскольку в командах INSERT, UPDATE, DELETE нет возможности указать порядок строк, то команда, затрагивающая несколько строк, приводит к ошибке:

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

```
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
```

А если запрос не вернет ни одной строки?

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

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

Переменные будут содержать неопределенные значения.

То же относится и командам INSERT, UPDATE, DELETE. Например:

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

```
NOTICE:  Изменили: (,)
DO
```

Иногда хочется быть уверенным, что в результате выборки получилась ровно одна строка: ни больше, ни меньше. В этом случае удобно воспользоваться фразой INTO STRICT:

```
=> 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
```

Как мы видели, команды INSERT, UPDATE, DELETE, затрагивающие несколько строк, приводят к ошибке. Фраза STRICT позволяет гарантировать, что строка будет ровно одна (а не ноль):

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

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

Явная проверка состояния

Другая возможность — проверять состояние последней выполненной SQL-команды:

- Команда GET DIAGNOSTICS позволяет получить количество затронутых строк (row_count);
- Предопределенная логическая переменная FOUND показывает, была ли затронута хотя бы одна строка.

```
=> 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
```

Заметьте: диагностика не позволяет обнаружить, что запросу соответствует нескольких строк, поскольку row_count возвращает единицу.

Табличные функции

Строки запроса

```
RETURN QUERY запрос;
```

Одна строка

```
RETURN NEXT выражение;      если нет выходных параметров
RETURN NEXT;                если есть выходные параметры
```

Особенности

строки добавляются к результату,
но выполнение функции не прекращается

команды можно выполнять несколько раз

результат не возвращается, пока функция не завершится

Чтобы создать на PL/pgSQL табличную функцию, нужно объявить ее как RETURNS SETOF или RETURNS TABLE (точно так же, как и для SQL).

Чтобы вернуть из функции множество строк, надо воспользоваться специальной конструкцией RETURNS QUERY *запрос*. Результат будет практически таким же, как в случае SQL-функции, содержащей этот запрос. Однако запрос из SQL-функции имеет шансы быть подставленным в объемлющий запрос, а в случае PL/pgSQL-функции это исключено.

Можно возвращать результат и построчно, используя конструкцию RETURN NEXT. Она похожа на обычный RETURN, но не прекращает выполнение функции, а добавляет возвращаемое значение в качестве очередной строки будущего результата. Команду RETURN NEXT (и RETURN QUERY тоже) можно вызывать несколько раз.

Окончательный результат будет возвращен только, когда выполнение функции полностью завершится (для этого можно использовать обычную команду RETURN). Иными словами, RETURN NEXT *не* работает как yield в функциях-генераторах современных языков.

https://postgrespro.ru/docs/postgresql/12/plpgsql-control-structures#PLPGSQL-STATEMENTS-RETURNING

Табличные функции

Пример табличной функции на 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 | Раз!
  2 | Два
(2 rows)
```

Другой вариант — возвращать значения построчно.

```
=> 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
--------------+------------
 Пн           |          1
 Вт           |          2
 Ср           |          3
 Чт           |          4
 Пт           |          5
 Сб           |          6
 Вс           |          7
(7 rows)
```

Почему функция объявлена как STABLE?

Потому что, хотя значение функции не зависит ни от параметров, ни от данных, оно тем не менее неявно зависит от текущей локали:

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

```
SET
```

```
=> 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)
```

Итоги

PL/pgSQL тесно интегрирован с SQL

- в процедурном коде можно выполнять запросы (оформленные как выражения или отдельные команды)
- в запросах можно использовать переменные
- можно получать результаты запросов и их статус

Нужно следить за неоднозначностями разрешения имен

Практика

1. Напишите функцию add_author для добавления новых авторов. Функция должна принимать три параметра (фамилия, имя, отчество) и возвращать идентификатор нового автора.

   Проверьте, что приложение позволяет добавлять авторов.

2. Напишите функцию buy_book для покупки книги. Функция принимает идентификатор книги и уменьшает количество таких книг на складе на единицу. Возвращаемое значение отсутствует.

   Проверьте, что в «Магазине» появилась возможность покупки книг.

1.

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

3.

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

Вы можете обратить внимание, что при покупке книг приложение позволяет «уйти в минус». Если бы количество книг хранилось в столбце, простым и хорошим решением было бы сделать ограничение CHECK. Но в нашем случае количество рассчитывается, и мы отложим написание проверки до темы «Триггеры».

1. Функция add_author

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

```
CREATE FUNCTION
```

2. Функция buy_book

```
=> CREATE OR REPLACE 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
```

Практика

Напишите игру, в которой сервер пытается угадать загаданное пользователем животное, задавая последовательные уточняющие вопросы, на которые можно отвечать «да» или «нет».

Если сервер предложил неправильный вариант, он запрашивает у пользователя имя животного и отличающий вопрос. Эта новая информация запоминается и используется в следующих играх.

1. Создайте таблицу для представления информации.
2. Придумайте интерфейс и реализуйте необходимые функции.
3. Проверьте реализацию.

Пример диалога (между людьми):

— Это млекопитающее?
— Это слон?
— Сдаюсь. Кто это?
— Как отличить кита от слона?

— Да.
— Нет.
— Кит.
— Он живет в воде.

1. Информацию удобно представить в виде двоичного дерева. Внутренние узлы хранят вопросы, листовые узлы — названия животных. Один из дочерних узлов соответствует ответу «да», другой — ответу «нет».

2. Между вызовами функций надо передавать информацию о том, на каком узле дерева мы остановились («контекст» диалога). Функции могут быть, например, такими:

- начать игру (нет входного контекста)

```
FUNCTION start_game(OUT context integer, OUT question text)
```

- продолжение игры (получаем ответ, выдаем следующий вопрос)

```
FUNCTION continue_game(
    INOUT context integer, IN answer boolean,
    OUT you_win boolean, OUT question text)
```

- завершение игры (внесение информации о новом животном)

```
FUNCTION end_game(
    IN context integer, IN name text, IN question text)
RETURNS void
```

1. Таблица

```
=> CREATE TABLE animals(
    id     integer PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    yes_id integer REFERENCES animals(id),
    no_id  integer REFERENCES animals(id),
    name   text
);
```

```
CREATE TABLE
```

```
=> INSERT INTO animals(name) VALUES
    ('млекопитающее'), ('слон'), ('черепаха');
```

```
INSERT 0 3
```

```
=> UPDATE animals SET yes_id = 2, no_id = 3 WHERE id = 1;
```

```
UPDATE 1
```

```
=> SELECT * FROM animals ORDER BY id;
```

```
 id | yes_id | no_id |     name
----+--------+-------+--------------
  1 |      2 |     3 | млекопитающее
  2 |        |       | слон
  3 |        |       | черепаха
(3 rows)
```

Первая строка считается корнем дерева.

2. Функции

```
=> CREATE FUNCTION start_game(
    OUT context integer,
    OUT question text
)
AS $$
DECLARE
    root_id CONSTANT integer := 1;
BEGIN
    SELECT id, name||'?'
    INTO context, question
    FROM animals
    WHERE id = root_id;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE FUNCTION continue_game(
    INOUT context integer,
    IN answer boolean,
    OUT you_win boolean,
    OUT question text
)
AS $$
DECLARE
    new_context integer;
BEGIN
    SELECT CASE WHEN answer THEN yes_id ELSE no_id END
    INTO new_context
    FROM animals
    WHERE id = context;

    IF new_context IS NULL THEN
        you_win := NOT answer;
        question := CASE
            WHEN you_win THEN 'Сдаюсь'
            ELSE 'Вы проиграли'
        END;
    ELSE
        SELECT id, null, name||'?'
        INTO context, you_win, question
        FROM animals
        WHERE id = new_context;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

```
=> CREATE FUNCTION end_game(
    IN context integer,
    IN name text,
    IN question text
) RETURNS void
AS $$
DECLARE
    new_animal_id integer;
    new_question_id integer;
BEGIN
    INSERT INTO animals(name) VALUES (name)
        RETURNING id INTO new_animal_id;
    INSERT INTO animals(name) VALUES (question)
        RETURNING id INTO new_question_id;
    UPDATE animals SET yes_id = new_question_id
    WHERE yes_id = context;
    UPDATE animals SET  no_id = new_question_id
    WHERE  no_id = context;
    UPDATE animals SET yes_id = new_animal_id, no_id = context
    WHERE id = new_question_id;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION
```

3. Пример сеанса игры

Загадываем слово «кит».

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

```
 context |    question
---------+----------------
       1 | млекопитающее?
(1 row)
```

```
=> SELECT * FROM continue_game(1,true);
```

```
 context | you_win | question
---------+---------+----------
       2 |         | слон?
(1 row)
```

```
=> SELECT * FROM continue_game(2,false);
```

```
 context | you_win | question
---------+---------+----------
       2 | t       | Сдаюсь
(1 row)
```

```
=> SELECT * FROM end_game(2,'кит','живет в воде');

 end_game
----------

(1 row)
```

Теперь в таблице:

```
=> SELECT * FROM animals ORDER BY id;

 id | yes_id | no_id |     name
----+--------+-------+---------------
  1 |      5 |     3 | млекопитающее
  2 |        |       | слон
  3 |        |       | черепаха
  4 |        |       | кит
  5 |      4 |     2 | живет в воде
(5 rows)
```

Снова загадали «кит».

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

 context |    question
---------+----------------
       1 | млекопитающее?
(1 row)
```

```
=> SELECT * FROM continue_game(1,true);

 context | you_win |   question
---------+---------+---------------
       5 |         | живет в воде?
(1 row)
```

```
=> SELECT * FROM continue_game(5,true);

 context | you_win | question
---------+---------+----------
       4 |         | кит?
(1 row)
```

```
=> SELECT * FROM continue_game(4,true);

 context | you_win |   question
---------+---------+--------------
       4 | f       | Вы проиграли
(1 row)
```