PLpgSQL versus PL/SQL

Pavel Stěhule

PgConf.Russia 2016

Pavel Stěhule

- Developer, designer, consultant, teacher, instructor
- PLpgSQL developer since 2005 (PostgreSQL 8.1)
  - variadic parameters, default parameters
  - `RETURN QUERY`, `CONTINUE`, `FOREACH SLICE`, `GET STACKED DIAGNOSTICS`, `ASSERT`
  - `USAGE` clause in `EXECUTE`
  - rich `RAISE` statement
  - *plpgsql_check*, *Orafce*
- functions: greatest, least, `format`, `string_agg`, `left`, `right`,
- \sf, \ef, \gset

PLpgSQL

```
CREATE OR REPLACE FUNCTION new_customer(name text, surname text)
RETURNS int AS $$
DECLARE uid int;
BEGIN
  IF NOT EXISTS(SELECT * FROM customers c
                  WHERE c.name = new_customer.name
                    AND c.surname = new_customer.surname)
  THEN
    INSERT INTO customers(name, surname)
       VALUES(new_customer.name, new_customer.surname
       RETURNING id INTO uid;
    RETURN uid;
  ELSE
    RAISE EXCEPTION "Customer exists already";
  END IF;
END;
$$ LANGUAGE plpgsql STRICT;
```

PLpgSQL

- ADA (PL/SQL) based language
- Algol like family: Pascal, Modula, ADA, Visual Basic, … - **verbose languages**
- Language is reduced - no I/O, packages, procedures
- Language is enhanced - SQL is part of language
- Reduce network overhead
- Helps with application decomposition
- Helps with security

PLpgSQL <=> PL/SQL

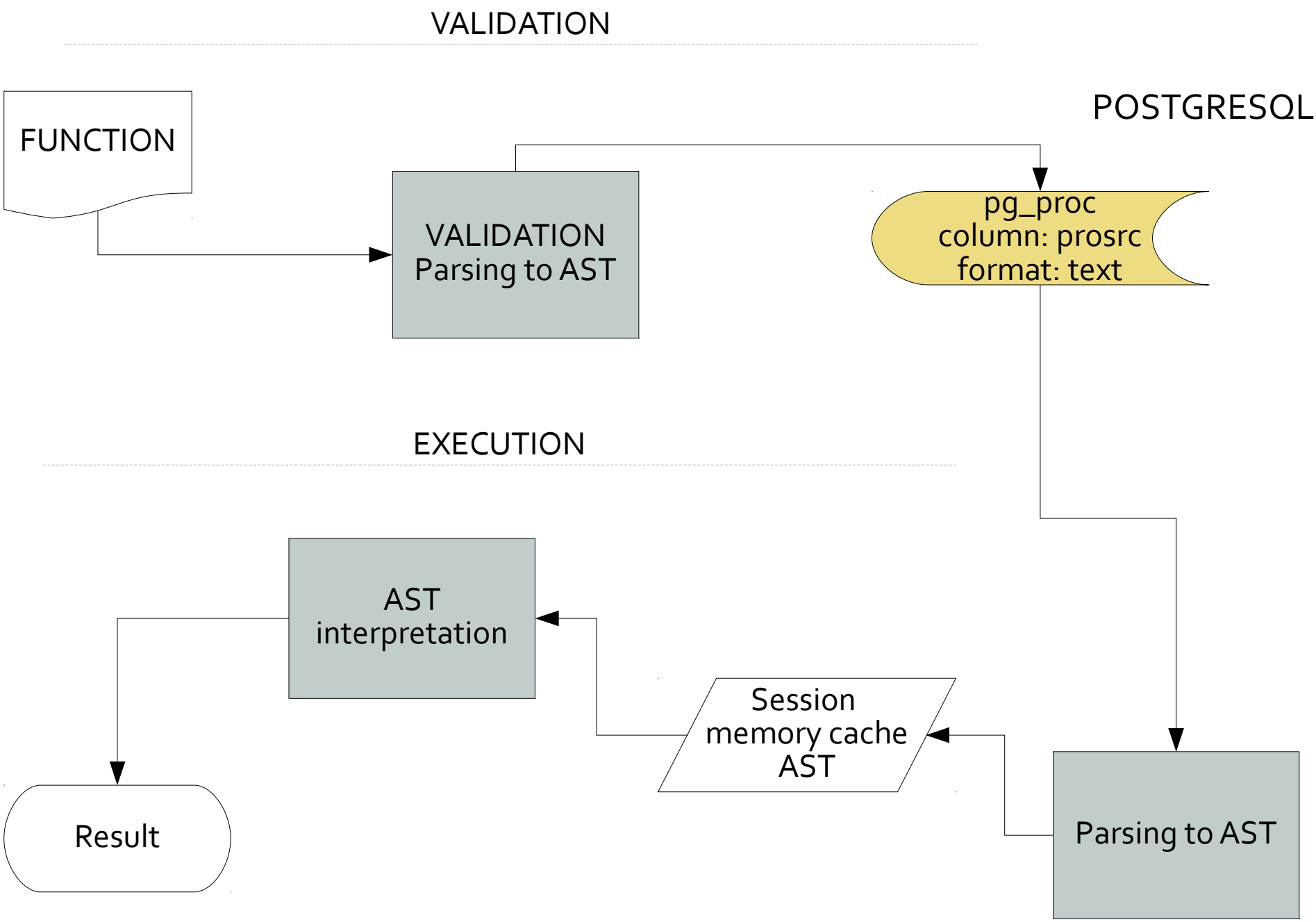
- Sometimes exact match
  - `FOR i IN 1 .. 10 LOOP`
- Sometimes partial match
  - `FOR r IN SELECT * FROM ...`
  - `EXECUTE ~~IMMEDIATE~~ '....'`
- Sometimes zero match
  - `dbms_output.put_line(...)`
  - `RAISE NOTICE '...'`
- Sometimes a default is opposite
  - `SECURITY INVOKER` (Postgres)
  - SECURITY DEFINER (Oracle, MSSQL, SQL/PSM)

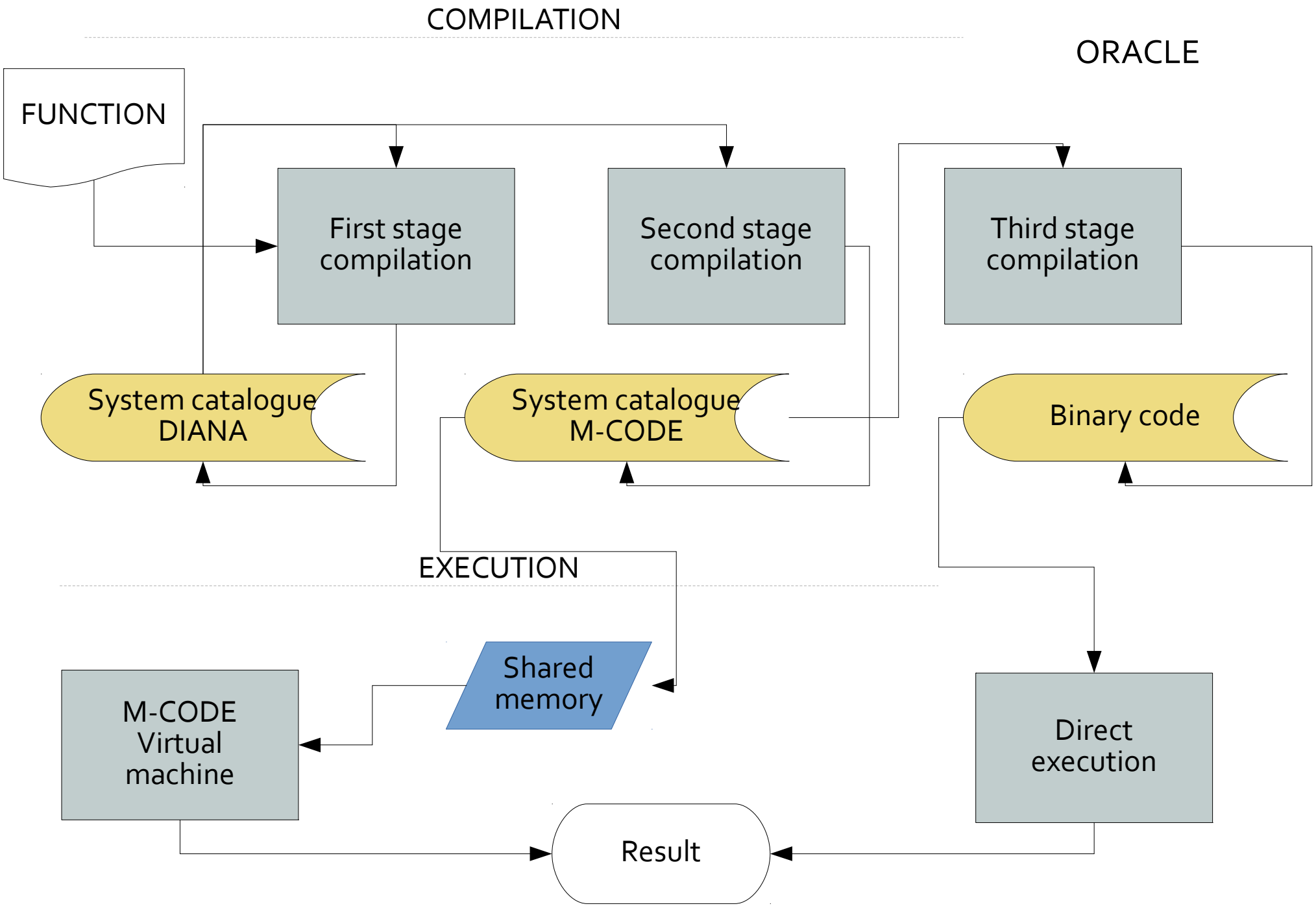
PLpgSQL <=> PL/SQL

- PLpgSQL looks like PL/SQL
- Originally primitive PL/SQL clone
- Different implementation
  - Bison parser, AST interpret, primitive SQL parsing, in process
- Now
  - Bison parser, AST interpret, smart integration with SQL parser/analyzer, in process

BASIC RULE

- **NEWER EDIT PLPGSQL WITH PGADMIN!!!**
- Use your preferred editor and edit file, deploy file
- use git ...
- Automatize by make, Makefile
- **Regress tests are important**


VALIDATION
POSTGRESQL
FUNCTION
VALIDATION
Parsing to AST
pg_proc
column: prosrc
format: text
EXECUTION
AST
interpretation
Session
memory cache
AST
Result
Parsing to AST


COMPILATION
ORACLE
FUNCTION
First stage compilation
Second stage compilation
Third stage compilation
System catalogue DIANA
System catalogue M-CODE
Binary code
EXECUTION
Shared memory
M-CODE Virtual machine
Direct execution
Result

Important design points

- **Simplicity!!!!**
- No compile (validation) dependencies
- Native (internal) stored procedures are supported.
  - code is stored in raw source code form (**pg_proc.prosrc**)
- Binary (external) stored procedures are supported.
  - the path to routine is stored (pg_proc.probin)
- Strong integration with SQL engine - SQL is same everywhere.

PLpgSQL

- Parser produces AST in local memory.
  - used as validator (> /dev/null)
- **Executor interprets AST in local memory/**
- Almost only local memory is used.
- AST is never serialized / read from IO.
- AST can be displayed, but nobody use it.
  - #option dump

No compile time dependencies

- Validation is fast (less access to system tables, processing only one object)
- Tools are simple (no dependencies)
- Some patterns are possible (local temp tables)

- But some bugs (typo) are detected too late
- (in runtime).
- *plpgsql_check* is necessary for larger code base

AST based interpret

- The code is simple and clean.
- **Only one stage in processing - parsing**
- Evaluation is effective - almost all code is executed by optimized C code.
- New features are implemented simply.

- Some statements are hard to implement.
  - PLpgSQL has not `GOTO` statement (no plan to fix it)
- Expressions are evaluated by SQL engine.
  - 100% compatible with PostgreSQL

In-process execution

- PostgreSQL uses one process per session.
- This process is used for SQL engine and PL runtime.

- No inter-process communication, no overhead
- Usually without problems
  - **TRUST languages are safe** - PLpgSQL
- *But possible dangerous in untrusted languages* - try set timeout in PLPerlu libraries and run operation slower 1sec. => segfault process => enforced server restart

Oracle PL/SQL

- **Based on complete ADA interpret (environment)**
  - with some compilation support
- Strong, but pretty complex
- Dependencies between objects
- **Massive libraries**
- Separate process from database engine

PL/SQL compiler

- When bottleneck are SQL statements, then speed of procedures are not significant.
- Some code doesn't use SQL - speed is important.
- Byte code interpretation without JIT is slow.
- Oracle fix - translate byte code “M-CODE” to C and compile (solution from pre JIT era).
  - significantly faster than M-CODE interpret
  - slower than native C-code

PostgreSQL reply

- **No PL/pgSQL compiler**
- There are lot of other fast PL with different speed characteristics (fast start, fast execution, fast string operations)
  - Python, Perl, Lua, Java,
- Nice and simple C - API
  - C is nice and simple language for short tasks
  - typical for string manipulation (see *Orafce*)

PL/pgSQL differences

- No OOP features
- Differently designed aggregates, exceptions
- Only local variables
  - possible to use server side custom setting variables (simple usage but slow)
  - possible to use PLPerl session variable
  - possible to use C extension (secure)
- **No packages - use schema instead**
- No collections - use arrays instead, or Perl hash
- No DBMS packages - but *Orafce* and **CPAN** (**untrusted**)
- No autonomous transactions - emulated by *dblink*

PLpgSQL to PL/SQL relation

- PLpgSQL is a clone of PL/SQL.
- Before EDB era there was a plan be compatible with PL/SQL.
- Leaved - PostgreSQL is not a Oracle clone now, usually we implement what we like and what is not big trap to Oracle developers.
- **Our strategy: PLpgSQL is simple to learn, simple to usage PL/SQL like language.** Who need 100% compatibility, use EDB.
- Some concepts from Oracle are not possible in Postgres (Oracle - one PL, Postgres multiple PL).
- Some concepts are complex or redundant.
  - Schema X packages
- PLpgSQL is first from more supported PL (not alone). Some features should be more generally designed.

Oracle PACKAGE

```
CREATE PACKAGE bonus AS
  PROCEDURE calc_bonus(uid int);
END

CREATE PACKAGE BODY bonus AS
  PROCEDURE calc_bonus(uid int) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('started');
  END;
END bonus;
```

PostgreSQL SCHEMA

```
DROP SCHEMA IF EXISTS bonus CASCADE;
CREATE SCHEMA bonus;
SET search_path TO bonus;

CREATE FUNCTION calc_bonus(uid int)
RETURNS void AS $$
BEGIN
  RAISE NOTICE 'started';
END;
$$ LANGUAGE plpgsql SET search_path = bonus;
```

Attention!

- PostgreSQL schema ≠ Oracle schema
- PostgreSQL schema ~ Oracle packages
  - no relation to user
  - no relation to storage

What we did?

- 8.4 `CASE`, rich exception, `VARIADIC` fce, `DEFAULT` params,
- 9.0 (2010) **detection ambiguous SQL and PLpgSQL** identifiers, naming & mixed notation
- 9.1 `FOREACH`
- 9.2 `GET STACKED DIAGNOSTICS`
- 9.3 event triggers, enhanced `GET STACKED DIAGNOSTICS`
- 9.4 enhanced `GET DIAGNOSTICS`,enhanced event triggers
- 9.5 `ASSERT`
- 9.6 valid context info for `RAISE EXCEPTION`, ??

What we want?
Reality

- Global temp tables (3 years)
  - more comfort for developers
  - less impact on performance (bloating pg_attribute)
- Autonomous transactions (3 years)
  - good for auditing, logs in tables, maintenance
- Static local variables (3 years)
  - can help with migration from Oracle packages

What we want?
Dreams

- Better work with dynamic complex types

  ```
  x.data[10].rec.{fieldname} = ...
  ```

- Basic scheduler, simple work-flow system based on notification handlers
- Procedures with some PL/SQL, T-SQL features
  - linear transactions (X nested transactions) - possible with autonomous transactions
  - multirecord sets

Doesn't do

- Doesn't migrate wrong badly designed code 1:1
  - don't supply client side code in procedures
    - interactivity
    - multilingual support, ..
  - don't supply communication server in procedures, database
    - own communication server (SOAP, REST) is more robust than database based communication
- **Relational database is not OOP database**
  - Entity relation diagram
  - Data flow diagram
  - No inheritance
- **All rules has exceptions!!!**

Do

- Use PL for data manipulation
- PL is designed for procedural principles (NOT OOP)
- Write procedures (not method)
- Oriented on business process implementation
- Separate layers
  - client: data input, data presentation
  - communication server:
  - database: data manipulation, data store

Do

- When your application is data oriented, (database centric)
  - verify your schema early
    - SQL queries should be readable
  - verify performance early
    - test performance important or frequented queries
- use *auto_explain*
  - `log_nested_statements = on`

plpgsql_check

- https://github.com/okbob/plpgsql_check
- two modes:
  - passive - LOAD 'plpgsql_check' (disabled by default)
  - active - plpgsql_check_function()

```
select * from plpgsql_check_function('f1()', fatal_errors := false);
                        plpgsql_check_function
------------------------------------------------------------------------
 error:42703:4:SQL statement:column "c" of relation "t1" does not exist
 Query: update t1 set c = 30
 --                     ^
 error:42P01:7:RAISE:missing FROM-clause entry for table "r"
 Query: SELECT r.c
 --             ^
 error:42601:7:RAISE:too few parameters specified for RAISE
(7 rows)
```

#option dump

```
Execution tree of successfully compiled PL/pgSQL function test(integer):
Function's data area:
entry 0: VAR $1                type int4 (typoid 23) atttypmod -1
entry 1: VAR found             type bool (typoid 16) atttypmod -1
entry 2: VAR x                 type int4 (typoid 23) atttypmod -1
DEFAULT 'SELECT a'
Function's statements:
4:BLOCK <<*unnamed*>>
5:  ASSIGN var 2 := 'SELECT x + a'
6:  RETURN variable 2
END -- *unnamed*
End of execution tree of function test(integer)
```