u cant seez
oh...
hai.
ICANHASCHEEZBURGER.COM

Developing a PostgreSQL Procedural Language

Joshua Tolley
End Point Corporation
eggyknap@gmail.com

I CAN HAZ PL'Z, PLZ?

SKEDJOOL

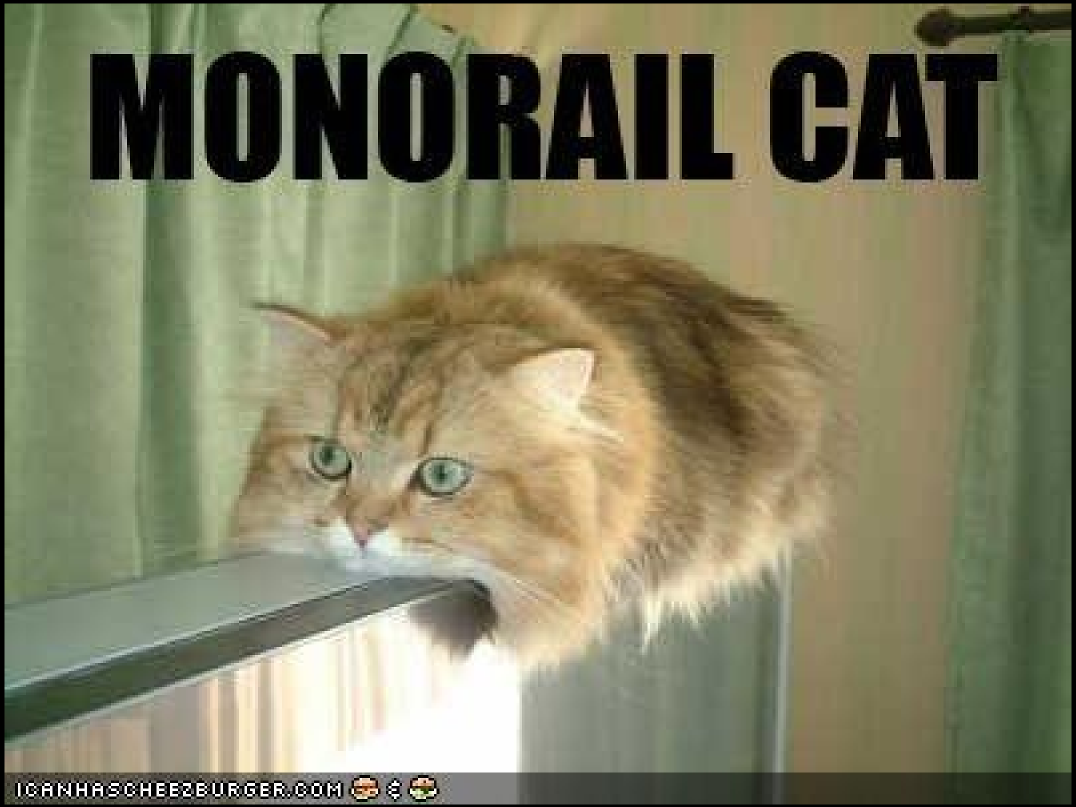
- What is PL/LOLCODE?
- How does it work
- How can I write one
  - ...and why might I ever want to?

WAT IZ [PL/]LOLCODE

- Began with LOLCATS:

A lolcat is an image combining a photograph, most frequently a cat, with a humorous and idiosyncratic caption in (often) broken English—a dialect which is known as "lolspeak", or "kitteh". The name "lolcat" is a compound word of "LOL" and "cat".[1] Another name is cat macro, being a type of image macro.[2] Lolcats are created for photo sharing imageboards and other internet forums. Lolcats are similar to other anthropomorphic animal-based image macros such as the O RLY? owl.

I made you a cookie...
but I eated it.


MONORAIL CAT
ICANHASCHEEZBURGER.COM

```
Additional TCP options:.
Starting moused:.
Starting background file system checks in 60 seconds.

Fri Aug  5 18:58:36 CEST 2005

FreeBSD/i386 (unixbox) (ttyv0)

login: root
Password:
Login incorrect
login: 6666666666666666666666666666tttt
```

dam it dam it dam it dam it
dam it dam it dam it dam it
ICANHASCHEEZBURGER.COM

WHENCE LOLCODE?

Adam Lindsay had a revelation. From Ceiling Cat:

“This is a love letter to very clever people who are slightly bored. I had no idea there were so many of us out there."

- FAQ, lolcode.com

HELLO, WORLD!

```
HAI
    CAN HAS STDIO?
    VISIBLE "HAI WORLD!"
KTHXBYE
```

A more complicated example...

```
HAI
    BTW Calculate pi, slowly, using Gregory-Leibniz series
    I HAS A PIADD ITZ 0.0
    I HAS A PISUB ITZ 0.0
    I HAS A ITR ITZ 0
    I HAS A ITRZ ITZ 20000
    I HAS A T1
    I HAS A T2

    IM IN YR LOOP
        T1 R QUOSHUNT OF 4.0 AN SUM OF 3.0 AN ITR
        T2 R QUOSHUNT OF 4.0 AN SUM OF 5.0 AN ITR
        PISUB R SUM OF PISUB AN T1
        PIADD R SUM OF PIADD AN T2
        ITR R SUM OF ITR AN 4.0
        BOTH SAEM ITR AN BIGGR OF ITR AN ITRZ, O RLY?
            YA RLY, GTFO
        OIC
    IM OUTTA YR LOOP
    FOUND YR SUM OF 4.0 AN DIFF OF PIADD AN PISUB
KTHXBYE
```

OBTW IT WERKZ

In PL/LOLCODE:

```
5432 eggyknap# SELECT
lol_find_pi_gregory_leibniz(1000);

 lol_find_pi_gregory_leibniz
-----------------------------
                    3.143589
(1 row)
```

SO WHY PL/LOLCODE?

I wrote PL/LOLCODE to …

SO WHY PL/LOLCODE?

I wrote PL/LOLCODE …

- To learn how

SO WHY PL/LOLCODE?

I wrote PL/LOLCODE …

- To learn how
- As a publicity stunt

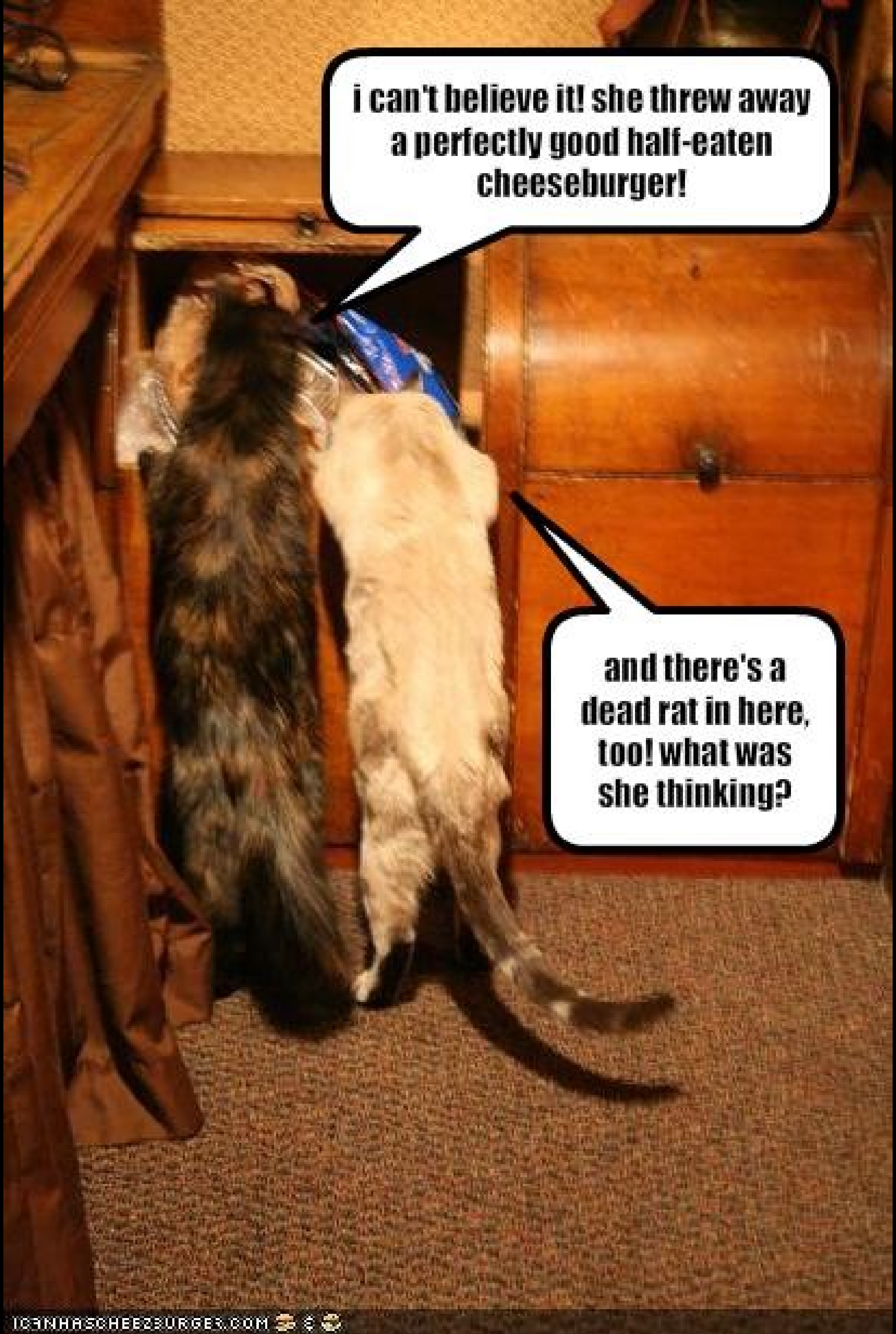
SO WHY PL/LOLCODE?

I wrote PL/LOLCODE …

- To learn how
- As a publicity stunt
- To achieve world domination

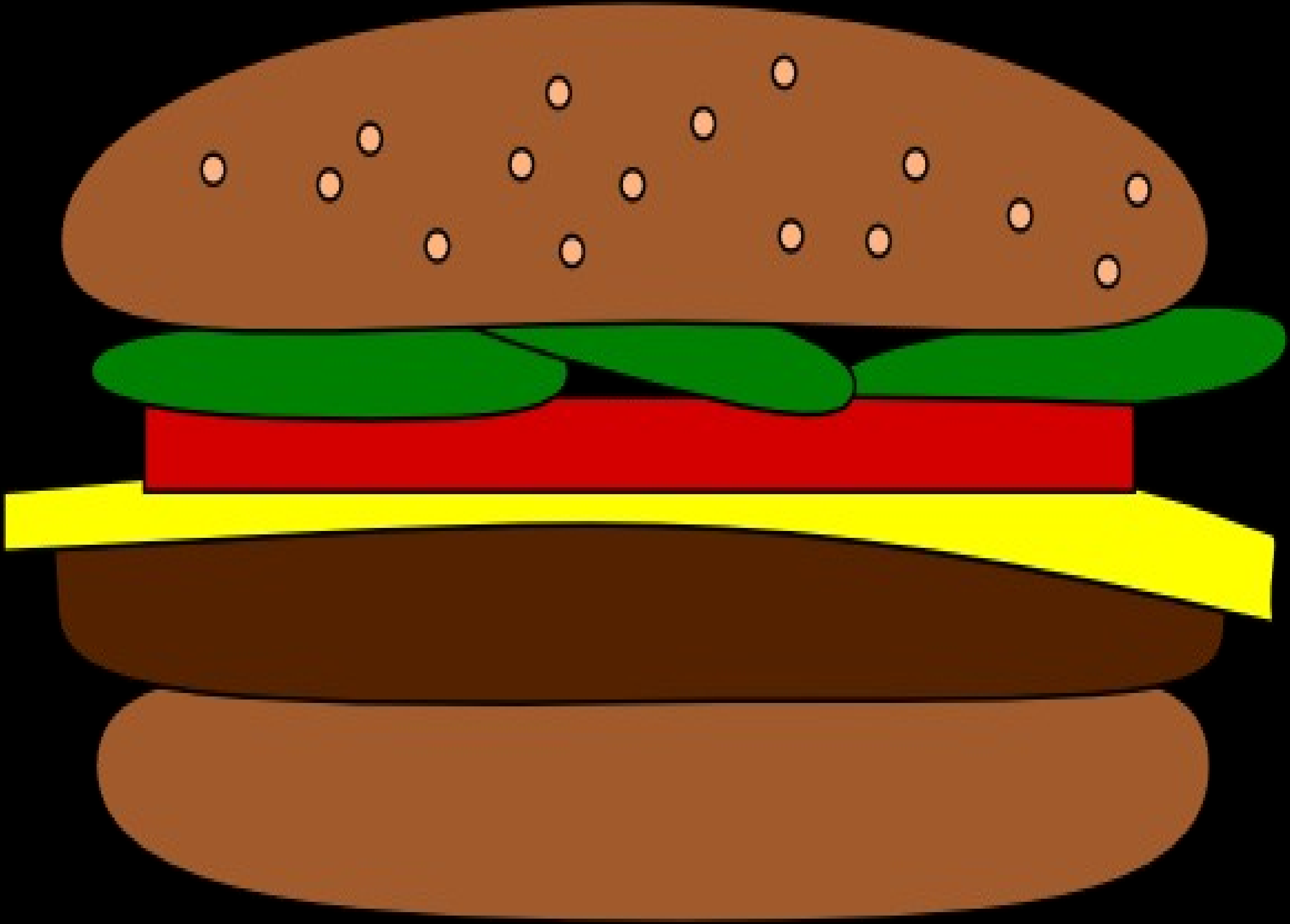
CHEEZBURGERS

... and now, a brief diversion ...


i can't believe it! she threw away a perfectly good half-eaten cheeseburger!
and there's a dead rat in here, too! what was she thinking?
ICANHASCHEEZBURGER.COM

Dis mi favret show
As teh cheezburger turns
ICANHASCHEEZBURGER.COM

I waitz here
for
cheezburgers
ICANHASCHEEZBURGER.COM



SPEEK LOLCODE

ANATUMMY OF A PL

FUNKSHUN CALL HANDLRZ

MAEK UN INTURPRETR

SPEEK LOLCODE

ANATUMMY OF A PL

FUNKSHUN CALL HANDLRZ

MAEK UN INTURPRETR

I CAN SPEEK LOLCODE

- Data types
  - NUMBR: Integer values
  - NUMBAR: Floating point values
  - YARN: String values
  - NOOB: Null values
  - TROOF: Boolean values (WIN / FAIL)

I CAN SPEEK LOLCODE

- Operators
  - Arithmetic
    - SUM OF x AN y, DIFF OF x AN y
    - PRODUKT OF x AN y, QUOSHUNT OF x AN y
    - MOD OF x AN y
    - BIGGR OF x AN y, SMALLR OF x AN y

I CAN SPEEK LOLCODE

- Operators
  - Boolean
    - BOTH OF x AN y, EITHER OF x AN y
    - WON OF x AN y
    - ALL OF x AN y [AN ...] MKAY
    - ANY OF x AN y [AN ...] MKAY
    - NOT x

I CAN SPEEK LOLCODE

- Operators
  - Comparison
    - BOTH SAEM x AN y
    - WIN iff x == y
    - DIFFRINT x AN y
    - FAIL iff x == y
  - Concatenation
    - SMOOSH x y z p d q ... MKAY
    - Concatenates infinitely many YARNs

I CAN SPEEK LOLCODE

- Operators
  - Casting
    - MAEK x A <type>
    - x IS NOW A <type>

COMPARISON

```
BOTH SAEM ANIMAL AN "CAT", O RLY?
    YA RLY, VISIBLE "J00 HAV A CAT"
    MEBBE BOTH SAEM ANIMAL AN "MAUS"
        VISIBLE "JOO HAV A MAUS? WTF?"
    NO WAI, VISIBLE "J00 SUX"
OIC
```

SWITCH/CASE

```
COLOR, WTF?
    OMG "R"
        VISIBLE "RED FISH"
        GTFO
    OMG "Y"
        VISIBLE "YELLOW FISH"
    OMG "G"
    OMG "B"
        VISIBLE "FISH HAS A FLAVOR"
        GTFO
    OMGWTF
        VISIBLE "FISH IS TRANSPARENT"
OIC
```

LOOPING

```
IM IN YR <label> <operation> YR
<variable>
  [TIL|WILE <expression>]
    <code block>
IM OUTTA YR <label>
```

IT

- “IT” is the default variable, if nothing else is specified
- Like Perl's $_ variable

PL/LOLCODE SPECIFIC...

- VISIBLE == RAISE NOTICE
  - Also, VISIBLE <level> “Message”
- FOUND YR <expression>
  - Returns a value
- GIMMEH <var> OUTTA DATABUKKIT “<query>”
  - Database interaction (SPI)

A PL/LOLCODE FUNCTION

```
CREATE FUNCTION lol_spi_test()
RETURNS TIMESTAMPTZ AS $$
    HAI
        I HAS A TIEM
        GIMMEH TIEM OUTTA DATABUKKIT
"SELECT NOW()"
        FOUND YR TIEM
    KTHXBYE
$$ LANGUAGE PLLOLCODE;
```

SPEEK LOLCODE

ANATUMMY OF A PL

FUNKSHUN CALL HANDLRZ

MAEK UN INTURPRETR

END point CORPORATION

ANATUMMY OF A PL

- Function call handler
  - Executes each stored procedure in a given language
  - Passed to CREATE LANGUAGE
  - Take LANGUAGE_HANDLER as a parameter
  - plpgsql_call_handler, plperl_call_handler, etc.
  - Languages C, SQL, and INTERNAL don't have handlers

ANATUMMY OF A PL

- Function call validator
  - Optional
  - Takes an OID argument
  - Invoked when stored procedures are created
  - Raise errors when procedures are invalid, otherwise return NULL

ANATUMMY OF A PL

- Trusted vs. untrusted
  - Trusted languages don't have effects outside of the database
  - Untrusted languages can do anything they want
  - PL/LOLCODE is trusted

ANATUMMY OF A PL

- What runs the language?
  - Interpreters can be built into the PL
    - PL/pgSQL, PL/LOLCODE
    - Can avoid data type conversion overhead
  - Load interpreter from a library
    - PL/Perl uses libperl
  - Something else
    - PL/J talks to a Java VM through RMI

SPEEK LOLCODE

ANATUMMY OF A PL

FUNKSHUN CALL HANDLRZ

MAEK UN INTURPRETR

FUNKSHUN CALL HANDLR

- Find argument types
- Find return type
- Get argument values
- Find function source
- Execute function
- Return proper result
- Triggers are a special case

FUNKSHUN CALL HANDLR

```
PG_MODULE_MAGIC;

Datum pl_lolcode_call_handler(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(pl_lolcode_call_handler);

Datum
pl_lolcode_call_handler(PG_FUNCTION_ARGS)
{
    /* Actual work goes here */
}
```

GET PG_PROC INFO

- Handler gets a FunctionCallInfo struct, fcinfo
- fcinfo contains the procedure's OID
- Use OID to get a HeapTuple object from pg_proc
- Use HeapTuple to get a Form_pg_proc struct, describing the procedure being called

GET PG_PROC INFO

```
Form_pg_proc procStruct;
HeapTuple procTup;

procTup = SearchSysCache(PROCOID,
ObjectIdGetDatum(fcinfo->flinfo->fn_oid), 0, 0, 0);

if (!HeapTupleIsValid(procTup)) elog(ERROR, "Cache
lookup failed for procedure %u", fcinfo->flinfo-
>fn_oid);

procStruct = (Form_pg_proc) GETSTRUCT(procTup);

ReleaseSysCache(procTup);
```

GET ARGS AND TYPES

```
for (i = 0; i < procStruct->pronargs; i++)
    {
        snprintf(arg_name, 255, "LOL%d", i+1);
        lolDeclareVar(arg_name);
        LOLifyDatum(fcinfo->arg[i], fcinfo-
>argnull[i], procStruct->proargtypes.values[i],
arg_name);
    }
```

LOLifyDatum() builds a PL/LOLCODE variable from type information and a string data value

GET RETURN TYPE

```
typeTup = SearchSysCache(TYPEOID,
ObjectIdGetDatum(procStruct->prorettype), 0, 0,
0);

if (!HeapTupleIsValid(typeTup)) elog(ERROR,
"Cache lookup failed for type %u", procStruct-
>prorettype);

typeStruct = (Form_pg_type) GETSTRUCT(typeTup);
resultTypeIOParam = getTypeIOParam(typeTup);

fmgr_info_cxt(typeStruct->typinput, &flinfo,
TopMemoryContext);
```

GET PROC SOURCE

```
procsrcdatum =
SysCacheGetAttr(PROCOID, procTup,
Anum_pg_proc_prosrc, &isnull);

if (isnull) elog(ERROR, "Function
source is null");

proc_source =
DatumGetCString(DirectFunctionCall1(
textout, procsrcdatum));
```

GET PROC SOURCE

Pass the source to the interpreter:

```
pllolcode_yy_scan_string(proc_source);
pllolcode_yyparse(NULL);
pllolcode_yylex_destroy();
```

Run the procedure (more on this later)

RETURN SOMETHING

```
if (returnTypeOID != VOIDOID) {
    if (returnVal->type == ident_NOOB) fcinfo->isnull =
true;
    else {
        if (returnTypeOID == BOOLOID)
            retval = InputFunctionCall(&flinfo,
lolVarGetTroof(returnVal) == lolWIN ? "T" : "F",
resultTypeIOParam, -1);
                else {
                    /* ... */
                    retval = InputFunctionCall(&flinfo,
rettmp, resultTypeIOParam, -1);
                }
        }
}
```

FMGR INTERFACE

- fmgr.c, fmgr.h define functions and structs used to call backend functions in PostgreSQL
- InputFunctionCall, OutputFunctionCall
  - Calls a previously-determined datatype input or output function
- DirectFunctionCall[1..9]
  - Call named functions with 1-9 arguments

SPEEK LOLCODE

ANATUMMY OF A PL

FUNKSHUN CALL HANDLRZ

MAEK UN INTURPRETR

MAEK UN INTERPRETR

- Interpreter consists of Parser and Executor
- Interpreter's job:
  - Take text string input
  - Derive some meaning from it
  - Build a “parse tree”
  - Run the parse tree

PARSER

- Don't write your parser from scratch
  - It's hard
  - Yours will be slow
  - Yours will be b0rken
  - Parser generators are widely available

PARSER GENERATORS

- Programmer supplies a language definition
- Tool translates definition into your language of choice
- cf. ANTLR, lex, flex, yacc, bison, Coco/R, ACCENT
- PL/LOLCODE uses flex + bison, because that's what PostgreSQL uses

FLEX + BISON

- Flex is a lexer
  - Divides input stream into tokens, called “lexemes”
  - Runs code when input matches constants or regexes
- Bison is a parser
  - Runs code when lexemes appear in syntactically meaningful ways

FLEX INPUT

```
/* Tokens (MAEK, etc.) defined in Bison cfg*/
MAEK         { return MAEK; }
"IS NOW A"   { return ISNOWA; }
A            { return A; }
TROOF        { return TROOFTYPE; }

[A-Za-z][A-Za-z0-9_]+  { return IDENTIFIER; }
[\+-]?[0-9]+
 {
   pllolcode_yylval.numbrVal = atoi(yytext);
   return NUMBR;
 }
```

FLEX STATE MACHINE

```
BTW                   { BEGIN LOL_LINECOMMENT; }
<LOL_LINECOMMENT>\n   { BEGIN 0; lineNumber++; }
<LOL_LINECOMMENT>.    {}

OBTW                  { BEGIN LOL_BLOCKCMT; }
<LOL_BLOCKCMT>TLDR    { BEGIN 0; }
<LOL_BLOCKCMT>\n      { lineNumber++; }
<LOL_BLOCKCMT>.       {}
```

FLEX → BISON

- Flex generates a stream of tokens
- Bison figures out what to do with various tokens when in meaningful order

BISON

```
/* Define tokens Flex will feed us */

%token HAI KTHXBYE EXPREND
%token YARN IDENTIFIER NUMBR NUMBAR
%token TROOFWIN TROOFFAIL TROOF
%token FOUNDYR IHASA ITZ R
...
```

BISON

Define grammar constructs:

```
%type <nodeList> lol_program lol_cmdblock
lol_orly_block lol_wtf_block
%type <node> lol_const lol_var lol_cmd
lol_xaryoprgroup lol_expression
%type <yarnVal> YARN
%type <yarnVal> IDENTIFIER
%type <numbrVal> NUMBR
%type <numbarVal> NUMBAR
%type <numbrVal> lol_binaryopr lol_unaryopr
lol_xaryopr lol_typename
```

BISON

Define each construct's meaning (~BNF):

```
lol_program:
    HAI EXPREND lol_cmdblock KTHXBYE EXPREND
                { yylval.nodeList = (lolNodeList
*) $3; }
    ;

lol_cmdblock:
    lol_cmdblock lol_cmd
                 { $$ = lolListAppend($1, $2); }
    | lol_cmd    { $$ = lolMakeList($1); }
    ;
```

BISON STACK

```
lol_typename:
    TROOFTYPE      { $$ = ident_TROOF; }
    | NUMBRTYPE    { $$ = ident_NUMBR; }
    | NUMBARTYPE   { $$ = ident_NUMBAR; }
    | YARNTYPE     { $$ = ident_YARN; }
    | NOOBTYPE     { $$ = ident_NOOB; }
    ;

/* $$ == “top of stack”. Each action pushes a
different constant onto the stack */
```

BISON STACK

What data type does the stack contain?

```
%union  {
     int numbrVal;
     double numbarVal;
     char *yarnVal;
     struct lolNodeList *nodeList;
     struct lolNode *node;
}
```

BISON STACK

What data type does the stack contain?

```
%type <nodeList> lol_program lol_cmdblock
%type <node> lol_const lol_var lol_expression
%type <yarnVal> YARN
%type <yarnVal> IDENTIFIER
%type <numbrVal> NUMBR
%type <numbarVal> NUMBAR
%type <numbrVal> lol_typename lol_unaryopr

/* Each language construct references a member of
the union */
```

PARSE TREE

- Some constructs just push items onto the stack
  - `TROOFTYPE  { $$ = ident_TROOF; }`
- Some consume items from the stack

  -
    ```
    FOUNDYR lol_expression
      { $$ = lolMakeNode(FOUNDYR, tmpnval,
      lolMakeList($2)); }
    ```
- Quiz: What type does lolMakeList() take?

PARSE TREE

- PL/LOLCODE uses a linked-list structure for the parse tree

```
struct lolNode {
    NodeType node_type;
    struct lolNode *next;
    /* list for sub-nodes */
    lolNodeList *list;
    NodeVal nodeVal;
};
```

PARSE TREE

- Each node can have:
  - A type (required)
  - A value (optional)
  - A list of child nodes (optional)
  - A “next” node (required, except for last node in the tree)

PARSE TREE

- Node types mostly match tokens from the lexer
- Executor handles nodes based on their types

```
switch (node->node_type) {
    case VISIBLE:
        visibleNode(node);
        break;
    case YARN:
        yarnNode(node);
        break;
 /* ...  */
```

PARSE TREE

Some node types are very simple:

```
void yarnNode(lolNode *node)
{
    lolSetVar("IT", ident_YARN, node->nodeVal);
}
```

PARSE TREE

... and some are not so simple ...

```
lolIdent
comparisonNode(lolNode *node, lolIdent a, lolIdent b)
{
    lolIdent result;
    double x, y;

    result.type = ident_TROOF;
    result.value.numbrVal = 0;
    if (a.type != b.type) {
        if ((a.type == ident_NUMBR && b.type == ident_NUMBAR) ||
            (a.type == ident_NUMBAR && b.type == ident_NUMBR)) {
            if (a.type == ident_NUMBR)
                x = NUMBR2NUMBAR(a.value.numbrVal);
            else x = a.value.numbarVal;
            if (b.type == ident_NUMBR)
                y = NUMBR2NUMBAR(b.value.numbrVal);
            else y = b.value.numbarVal;
            result.value.numbrVal = ( x == y ) ? 1 : 0;
        }
    }
    else switch (a.type) {
        case ident_TROOF:
            if ((a.value.numbrVal > 0 && b.value.numbrVal > 0) || (a.value.numbrVal == 0 && b.value.numbrVal == 0))
                result.value.numbrVal = 1;
            break;
        case ident_NUMBR:
            if (a.value.numbrVal == b.value.numbrVal)
                result.value.numbrVal = 1;
            break;
        case ident_NOOB:
            break;
        case ident_NUMBAR:
            if (a.value.numbarVal == b.value.numbarVal)
                result.value.numbrVal = 1;
            break;
        case ident_YARN:
            result.value.numbrVal = (strcmp(b.value.yarnVal, a.value.yarnVal) == 0 ? 1 : 0);
            break;
    }
    if (node->node_type == DIFFRINT)
        result.value.numbrVal = (result.value.numbrVal == 1) ? 0 : 1;
    return result;
}
```

PARSE TREE

A few examples...

EXAMPLE

- VISIBLE “HAI, EVRYBUDDY!”
  - The VISIBLE commands creates one node of type VISIBLE
  - The “HAI, EVRYBUDDY” becomes a YARN node
  - The YARN node is a child of the VISIBLE node

YARN NODE

Created like this in Bison:

```
lol_const:
    YARN { tmpnval.yarnVal = pstrdup($1);
      $$ = lolMakeNode(YARN, tmpnval, NULL); }
```

Node logic:

```
void yarnNode(lolNode *node)
{
    lolSetVar("IT", ident_YARN, node->nodeVal);
}
```

VISIBLE NODE

Created in Bison like this:

```
    | VISIBLE lol_expression
         { tmpnval.numbrVal = NOTICE;
           $$ = lolMakeNode(VISIBLE, tmpnval,
lolMakeList($2)); }
```

Node logic:

```
void visibleNode(lolNode *node)
{
    executeList(node->list);
    elog(node->nodeVal.numbrVal, "%s",
      lolVarGetString(lolGetVar("IT"), false));
}
```

ANOTHER EXAMPLE

```
BOTH SAEM ITR AN BIGGR OF ITR AN ITERASHUNZ,
O RLY?
    YA RLY, VISIBLE “BIGGR!!1”
    NO WAI, VISIBLE “J00 SUX”
OIC
```

ANOTHER EXAMPLE

- This creates two nodes in the main parse tree
  - BOTHSAEM node
    - The values to be compared are stored as two nodes in the BOTHSAEM node's child list
    - The comparison result goes into the IT variable
  - ORLY node
    - Each sub-expression (YA RLY, NO WAI, MEBBE) is a node in the ORLY node's child list
    - The VISIBLE nodes are children of the YARLY and NOWAI nodes

SPI EXAMPLE

- SPI == Server Programming Interface
- Allows issuing SQL queries from the backend
- Note that connecting to SPI puts you in a new memory context

SPI EXAMPLE

```
/* Execute a query */

res = SPI_exec(lolVarGetString(IT, false), 0);
```

SPI EXAMPLE

```
switch (res) {
    case SPI_OK_SELECT:
        if (SPI_processed < 1) {
            elog(DEBUG5, "PL/LOLCODE SPI: No rows
returned");
            return;
        }
        SPIval = SPI_getvalue(SPI_tuptable-
>vals[0], SPI_tuptable->tupdesc, 1);
        LOLifyString(SPIval,
SPI_gettypeid(SPI_tuptable->tupdesc, 1), node-
>nodeVal.yarnVal);
        break;
```

SPI EXAMPLE

```
        case SPI_ERROR_ARGUMENT:
            elog(ERROR, "SPI_execute returned
SPI_ERROR_ARGUMENT. Please provide a proper query
to retrieve.");
            break;
        case SPI_ERROR_COPY:
            elog(ERROR, "PL/LOLCODE can't copy to
STDOUT or from STDIN");
            break;
        case SPI_ERROR_TRANSACTION:
   /* … lots of error conditions ... */
```

PERSISTENT PARSE TREE

- Parsing is slow
- Parsing should give the same result each time
    - If it doesn't, you've probably got problems
- Store the parse tree the first time it's built, and re-use it later

PERSISTENT PARSE TREE

- PostgreSQL features a built-in hash table mechanism
- Define structs for values and keys
- Easy searching and adding of values
- Pay attention to memory context
  - The parse tree must stay in memory a long time
  - Quiz: what's a PostgreSQL memory context?

PERSISTENT PARSE TREE

- Gotchas
  - User might modify a procedure, changing arguments or code
  - Key the hash table on more than just procedure OID
  - Otherwise you might run an old version of the procedure
  - PL/LOLCODE isn't quite that smart yet

PERSISTENT PARSE TREE

```
/* Hash table key and value structs */
typedef struct pl_lolcode_hashkey {
    Oid funcoid;
    bool isTrigger;
    Oid trigrelOid;
    Oid argtypes[FUNC_MAX_ARGS];
} pl_lolcode_HashKey;

typedef struct pl_lolcode_hashent {
    pl_lolcode_HashKey key;
    lolProgram *program;
} pl_lolcode_HashEnt;
```

PERSISTENT PARSE TREE

```
/* Searching the hash table */
MemSet(&key, 0, sizeof(pl_lolcode_HashKey));
key.funcoid = funcoid;
key.isTrigger = false;
key.trigrelOid = 0;
memcpy(key.argtypes, argtypes,
          sizeof(Oid) * numargs);
entry =
  (pl_lolcode_HashEnt *)
    hash_search(pl_lolcode_HashTable,
      (void*) &key, HASH_FIND, NULL);
```

OTHR OPTIMIZAYSHUNZ

- PL/pgSQL keeps its variables as Datums
  - No type conversion necessary
- PL/pgSQL keeps variables in an array, and determines the position of each variable in the array at compile time
  - No searching of a table of variables is necessary
- PL/LOLCODE does neither of these, yet

DEVELOPMENT METHODS

TOOLZ

- A good editor
  - vim
- gdb
  - Debugger
  - Available everywhere
  - Full featured, if you don't mind a CLI

TOOLZ

- ctags
  - Builds a “tags” file from a set of source
  - Allows easy browsing through large projects
- cscope
  - Similar to ctags
  - Browse through projects with symbol names, strings, file names, etc.

QWESTSHUNZ?