P2 - A PERL11 PROJECT

```
5 + 6 = 11
```

```
http://perl11.org/p2/
```

p5p6 reunification summit preparing YAPC::EU 12


PIMC

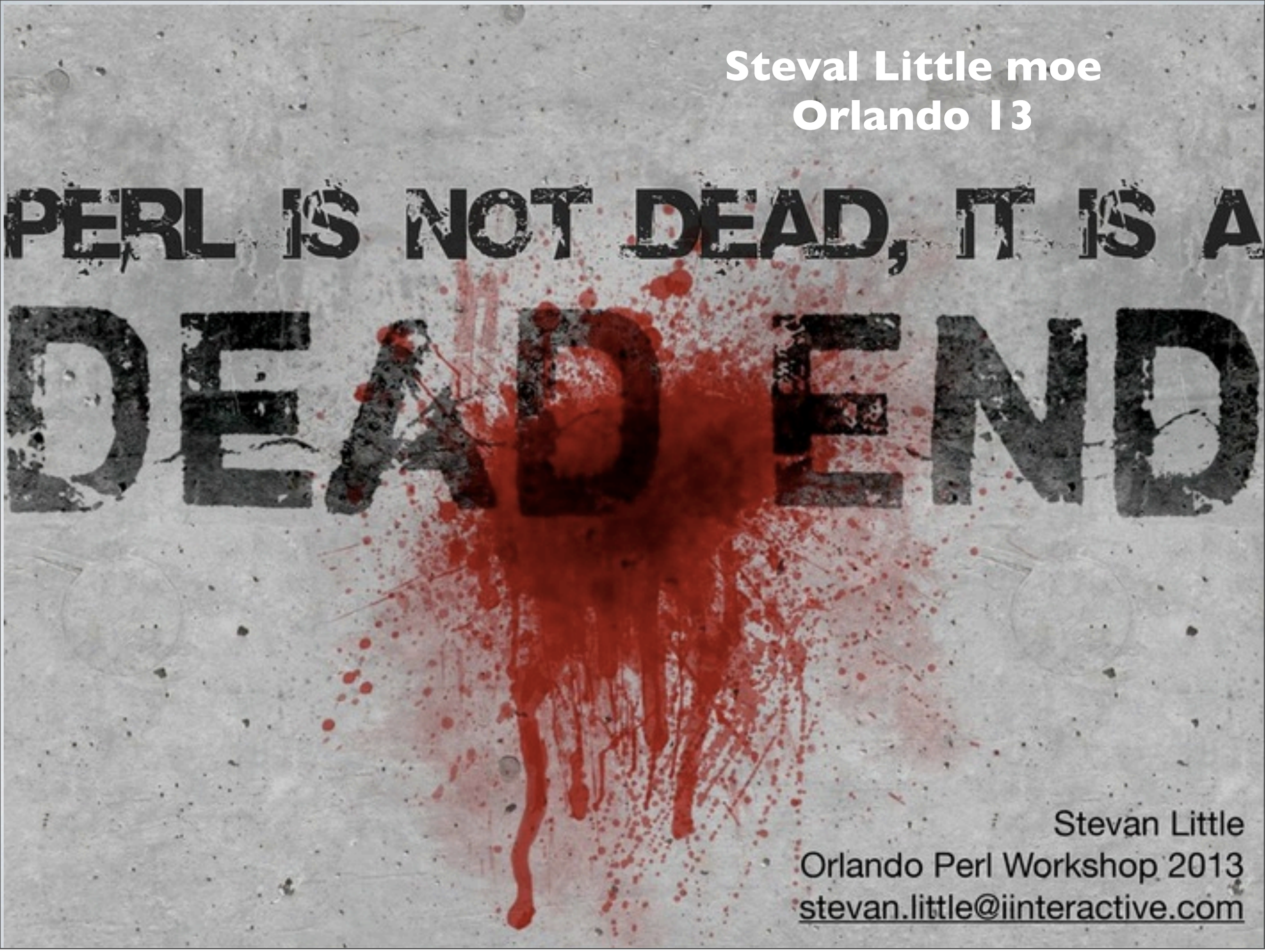
Will Braswell

Austin 2012

perl11 - preparing YAPC::US 13

Ingy döt net

Steval Little moe
Orlando 13



PERL8.ORG

pugs in scala - moe


IN GIFT SHOP
Deer
$1,500.


271 TO 280
#273


Baby Boy

perl11

Pluggable perl5 (+6)
プラガブル

1 **Parser** -> AST
パーザー

2 **Compiler** AST -> ops
コンパイラ

3 **VM** - Execute ops
を実行する

PARSER パーザー

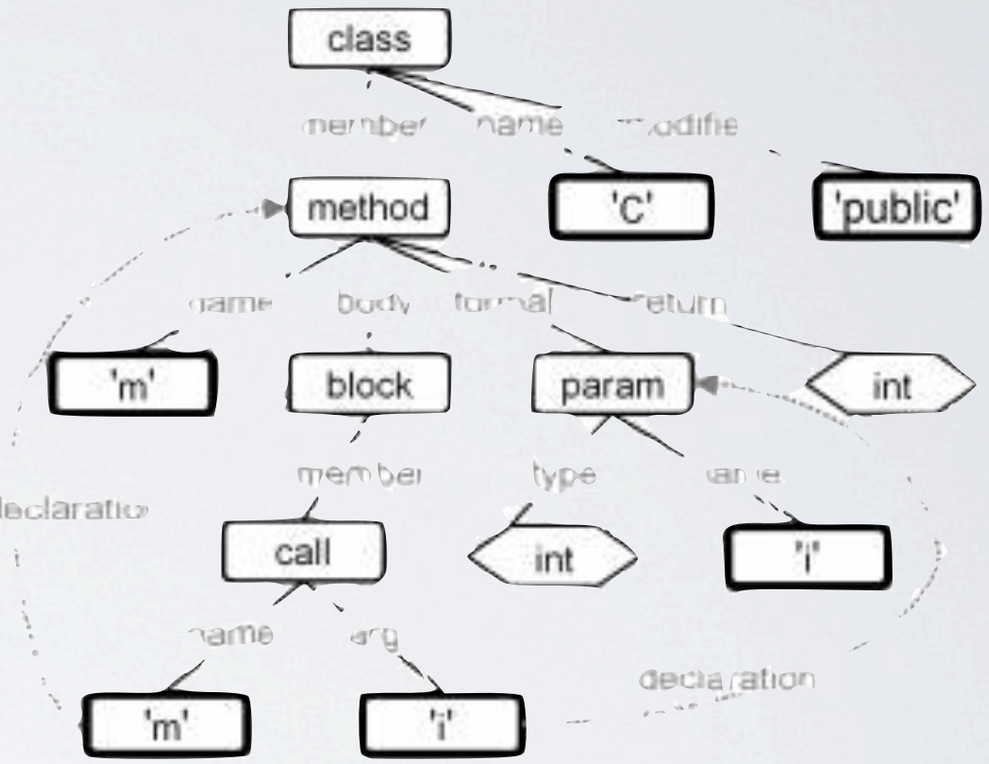
YACC *(bottom up)*

PEG / packrat *(top down)*

Marpa, ANTLR, ragel...

PGE, parsec, maru...

Handwritten



COMPILER コンパイラ

- AST -> ops linearization and optimizations 線形化と最適化
- Data Structures: native vs library
  データ構造 ネイティブ vs ライブラリ
- emit bytecode or jit. バイトコードかjitを実行
  => pluggable プラガブル : c, native, jvm, js

Data structures: native vs library tradeoff

VM (S)

Compile & execute compiled code コンパイル & コンパイルされたコードを実行

Bytecode or JIT バイトコードか JIT

Call-out/in native libs ネイティブのライブラリのc

Debugging support, reflection デバッギングサポート，リフレクション

ABI extensions, fast callbacks

DESIGN PRINCIPLES 設計方針

Frequent case

頻繁なケース

- Math 計算
- Conditionals 条件文
- Function calls 関数呼び出し
- Method dispatch メソッドディスパッチ
- Local variables ローカル変数
- Strings, build + compare
  文字列, ビルド + 比較
- Memory allocation メモリ割当
- Default arguments デフォルトの引数

Not

頻繁でないケース

- New methods 新しいメソッド
- Creation of classes クラスの作成
- Deep scoping situations
  深いスコープを使う状況
- Change inheritance tree
  継承ツリーの変更
- Global variables グローバル変数
- Eval
- Code allocation コードの割当
- Named arguments 名前付きの引数

LEARN FROM THE GOOD

良いものから学ぶ

- 30MB static C++ libs for **LLVM** just for a **JIT**?
  JITだけのためのLLVMに30MBの静的なライブラリ?

1236 loc, 87K

- 1GB of ugly junk for a JVM/.NET with huge startup overhead? Safe but not practical
  巨大な開始時のオーバーヘッドを伴う,JVM/.NETのため1GBのの醜いガラクタ? 安全だけど,実践的じゃない

- Java's main competitor の主な競争相手: Lucent Inferno OS/Limbo/**Dis** VM

- All "good" VMs use their approach: GC, register based, three-address coding, tagged small data, word-size ops
  全ての "良い" VMはそれらのアプローチを使っている: GC, レジスタベース, 3番地コーディング, タグ付けされた小さなデータ, ワードサイズops

Our JIT: 1236 loc, 87K

LEARN FROM THE GOOD

良いものから学ぶ

- 30MB static C++ libs for **LLVM** just for a **JIT**? 1236 lines, 87K
  JITだけのためのLLVMに30MBの静的なライブラリ?
- 1GB of ugly junk for a JVM/.NET with huge startup overhead? Safe but not practical
  巨大な開始時のオーバーヘッドを伴う,JVM/.NETのため1GBのの醜いガラクタ? 安全だけど,実践的じゃない
- Java's main competitor の主な競争相手: Lucent Inferno OS/Limbo/**Dis** VM
- All "good" VMs use their approach: GC, register based, three-address coding, tagged small data, word-size ops
  全ての "良い" VMはそれらのアプローチを使っている: GC, レジスタベース, 3番地コーディング, タグ付けされた小さなデータ, ワードサイズops

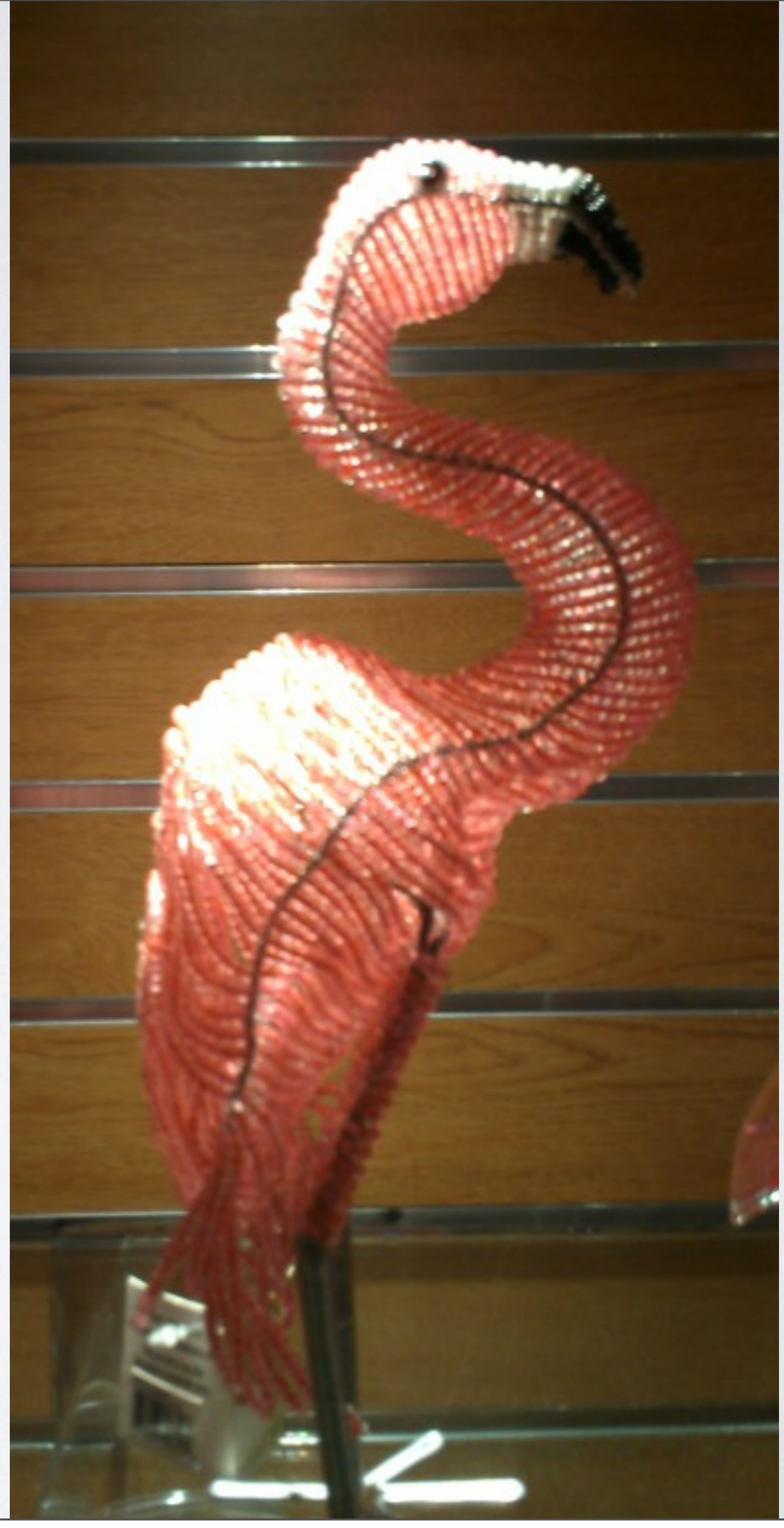
Our JIT: 1236 loc, 87K

POTION

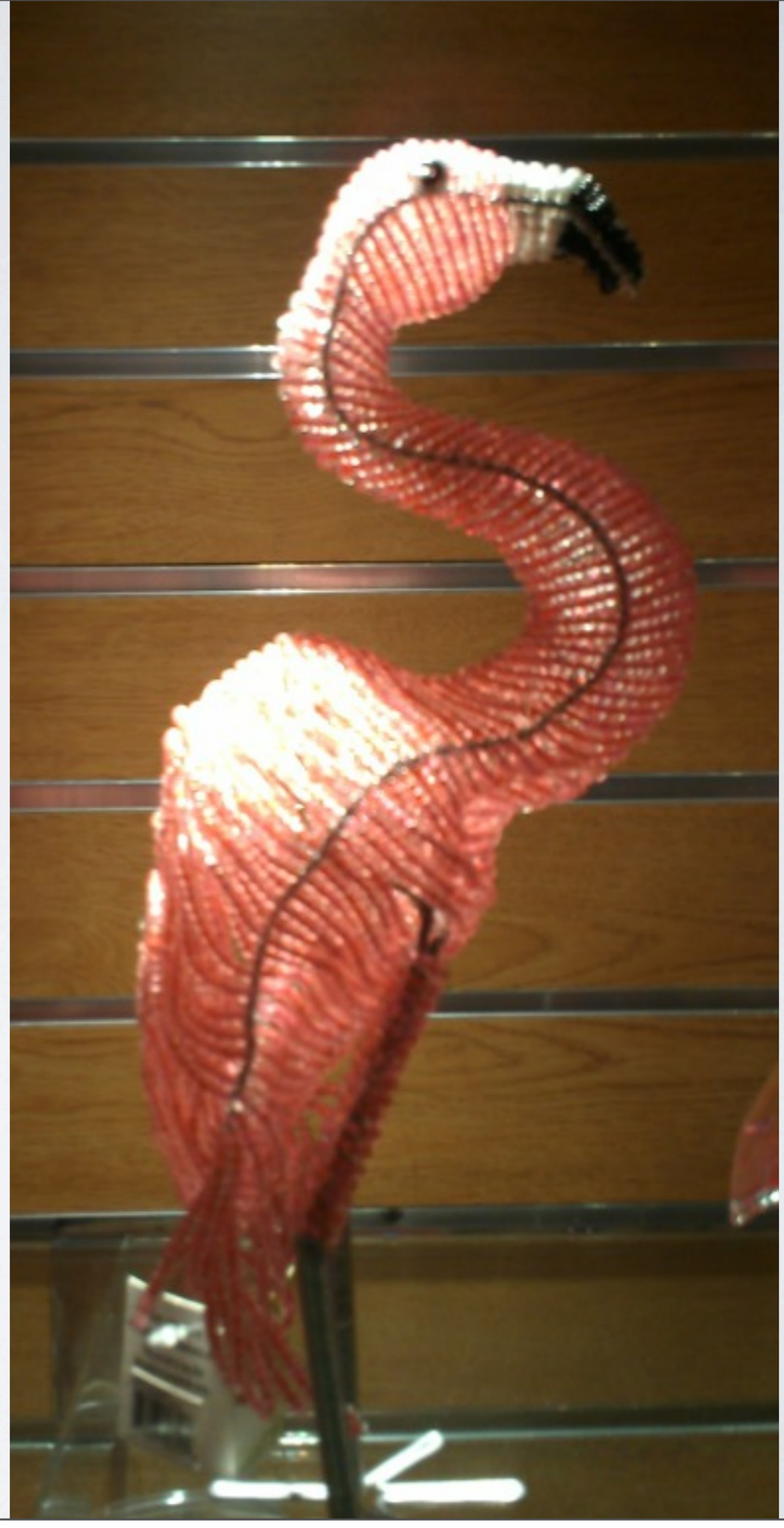
- why the lucky stiff -
  famous ruby eclectic. disappeared
  有名なrubyの折衷主義者,消えてしまった
- stack-based expressive, with data (lisp)
- lua VM
- io + soda objmodel のオブジェクトモデル (smalltalk based ベース)
- GC Cheney two-finger loop from QISH
  からのケニーの 2-finger ループ
- JIT self-written, very elegant JIT 自分で書いた,とても洗練されている

io: steven dekorte
soda: ian piumarta

potion



```
     .ooo
      'OOOo
~ p ooOOOo 2 ~
     .OOO
      oO       %% a fast perl5
        Oo
       'O
         `
        (o)
   ___/ /
  /`    \
 /v^    `   ,
(...v/v^/'
 \../::/
  \/::/
```



play

Premiering at

RubyConf

Denver, Colorado
November 1–3, 2012

featuring music and art by
Why The Lucky Stiff

POTION / P2

- common number interface 共通の数のインターフェース

POTION / P2

IV or NV, possibly auotmatic bignums

- common number interface - integer, double, bignums
- common hash/array interface 共通のハッシュ/配列のインターフェース

tables and tuples, interchangable (i.e. casting)

- IV or NV, possibly automatic bignums with CPU-specific overflow checks
- tables and tuples interchangable (i.e. automatic casting)

POTION / P2

- common number interface

共通の数のインターフェース

- common hash/array interface

テーブルとタプル,交換可能に(つまり自動的なキャスト)

```
%new = @a = %h
```

tables and tuples, interchangable (i.e. casting)

- IV or NV, possibly automatic bignums with CPU-specific overflow checks
- tables and tuples interchangable (i.e. automatic casting)

POTION / P2

- common number interface 共通の数のインターフェース
- common hash/array interface 共通のハッシュ/配列のインターフェース
- everything is an object, every object is a word
  全てはオブジェクト,全てのオブジェクトはワード

data, variables, functions, blocks, class, types, metaclasses, ...

POTION / P2

- common number interface 共通の数のインターフェース
- common hash/array interface 共通のハッシュ/配列のインターフェース
- everything is an object, every object is a word

  全てはオブジェクト,全てのオブジェクトはワード

  data, variables, functions, blocks, class, types, metaclasses, ...

  データ,変数,関数,ブロック,クラス,型,メタクラス, ...

functions, variables, blocks, class, types, metaclasses.

POTION / P2

- common number interface 共通の数のインターフェース
- common hash/array interface 共通のハッシュ/配列のインターフェース
- everything is an object, every object is a word
  全てはオブジェクト,全てのオブジェクトはワード
- every op is a word 全ての `op` はワード

PARSER パーザー

- peg, enhanced to greg　peg, gregに強化された
- Syntax tree of PNSource objects (max 3 nodes)
  PNSourceオブジェクトの構文木(最大3ノード)

```
$a = if (0) { 12 }
elsif (1) { 14 }
else { 16 }
```

```
ifstmt = IF e:ifexpr s:block - !"els"  { $$ = PN_OP(AST_AND, e, s) }
   | IF e:ifexpr s1:block -
      { $$ = e = PN_AST3(MSG, PN_if, PN_AST(LIST, PN_TUP(e)), s1) }
    (ELSIF e1:ifexpr f:block -
      { $$ = e = PN_PUSH(PN_TUPIF(e), PN_AST3(MSG, PN_elsif, PN_AST(LIST, PN_TUP(e1)), f)) } )*
    (ELSE s2:block
      { $$ = PN_PUSH(PN_TUPIF(e), PN_AST3(MSG, PN_else, PN_NIL, s2)) } )?

ifexpr = '(' - expr - ')' -
```

```
(assign (msg ("$a")
  expr (msg ("if" list (expr (value (0))) block (expr (value (12)))),
  msg ("elsif" list (expr (value (1))) block (expr (value (14)))),
  msg ("else" undef block (expr (value (16))))))
```

COMPILER コンパイラ

```
(assign (msg ("$a")
  expr (msg ("if" list (expr (value (0))) block (expr (value (12)))),
  msg ("elsif" list (expr (value (1))) block (expr (value (14)))),
  msg ("else" undef block (expr (value (16))))))
```

```
-- compiled --
; function definition: 0x1059ba7d8; 56 bytes
; () 3 registers
.local $a ; 0
[ 1] loadpn   1 1   ; 0
[ 2] notjmp   1 1   ; to 4
[ 3] loadpn   0 25  ; 12
[ 4] testjmp  1 3   ; to 8
[ 5] loadpn   1 3   ; 1
[ 6] notjmp   1 1   ; to 8
[ 7] loadpn   0 29  ; 14
[ 8] testjmp  1 1   ; to 10
[ 9] loadpn   0 33  ; 16
[10] self     1
[11] getlocal 2 0   ; $a
[12] call     0 2
[13] setlocal 0 0   ; $a
[14] return   0
; function end
```

COMPILER コンパイラ

```
(assign (msg ("$a")
  expr (msg ("if" list (expr (value (0))) block (expr (value (12)))),
  msg ("elsif" list (expr (value (1))) block (expr (value (14)))),
  msg ("else" undef block (expr (value (16))))))
```

```
-- compiled --
; function definition: 0x1059ba7d8; 56 bytes
; () 3 registers
.local $a ; 0
[ 1] loadpn   1 1   ; 0
[ 2] notjmp   1 1   ; to 4
[ 3] loadpn   0 25  ; 12
[ 4] testjmp  1 3   ; to 8
[ 5] loadpn   1 3   ; 1
[ 6] notjmp   1 1   ; to 8
[ 7] loadpn   0 29  ; 14
[ 8] testjmp  1 1   ; to 10
[ 9] loadpn   0 33  ; 16
[10] self     1
[11] getlocal 2 0   ; $a
[12] call     0 2
[13] setlocal 0 0   ; $a
[14] return   0
; function end
```

if is no *keyword*, just a **msg** on a **list** with a **block**. i.e. method on a list with a block argument.

`if` はキーワードではなく ただのブロック付きのリストのメッセージ． つまり,ブロックを引数に取る,リストのメソッド

COMPILER コンパイラ

```
(assign (msg ("$a")
  expr (msg ("if" list (expr (value (0))) block (expr (value (12)))),
  msg ("elsif" list (expr (value (1))) block (expr (value (14)))),
  msg ("else" undef block (expr (value (16))))))
```

```
-- compiled --
; function definition: 0x1059ba7d8; 56 bytes
; () 3 registers
.local $a ; 0
[ 1] loadpn   1 1   ; 0
[ 2] notjmp   1 1   ; to 4
[ 3] loadpn   0 25  ; 12
[ 4] testjmp  1 3   ; to 8
[ 5] loadpn   1 3   ; 1
[ 6] notjmp   1 1   ; to 8
[ 7] loadpn   0 29  ; 14
[ 8] testjmp  1 1   ; to 10
[ 9] loadpn   0 33  ; 16
[10] self     1
[11] getlocal 2 0   ; $a
[12] call     0 2
[13] setlocal 0 0   ; $a
[14] return   0
; function end
```

constant folding 定数折畳

if (value (0)) -> notjmp
elseif (value (1)) -> testjmp

if is no *keyword*, just a **msg** on a **list** with a **block**. i.e. method on a list with a block argument.

COMPILER

- Some control constructs are not parser special.

  いくつかの制御コンストラクタはパーザーに特別なものではない

  Expanded by the compiler, like a macro
  マクロのようにコンパイラによって拡張される

- Macros are compile-time parser extensions, no parser keywords -> extendable
  マクロはコンパイル時のパーザーの拡張であり,パーザのキーワードではない -> 拡張可能

- Most perl-level ops are just methods on objects
  perlレベルのopsの多くは,オブジェクトのただのメソッド.

- Compiler is extendable. コンパイラは拡張可能.

  --compile=c,opts calls the external compile-**c** library
  オプションは外部のコンパイル済みのcライブラリをロードして,呼び出す

- No optimizations yet (constant folding, dead code, tail call, ...)

VM

- Everything is an object, every object is a function (lambda)
  全てはオブジェクト,全てのオブジェクトは関数
- Every variable is a function, reacts to methods. (get, set, string, ...)
  全ての変数は関数,メソッドに反応する(get,set,文字列 ...)
- Every block is a function, with lexical scoped variables and env
  全てのブロックは関数で,レキシカルスコープの変数と環境変数を取る
- Every call is a method call, even on nil (undef) or any (UNIVERSAL, main::).
  全ての呼び出しはメソッド呼び出し,nilでもなんで(UNIVERSAL, main::)あっても

  functions are methods on the global obj

.

main:: if in main lexical scope. UNIVERSAL in class libraries, not scoped from main::

MOP

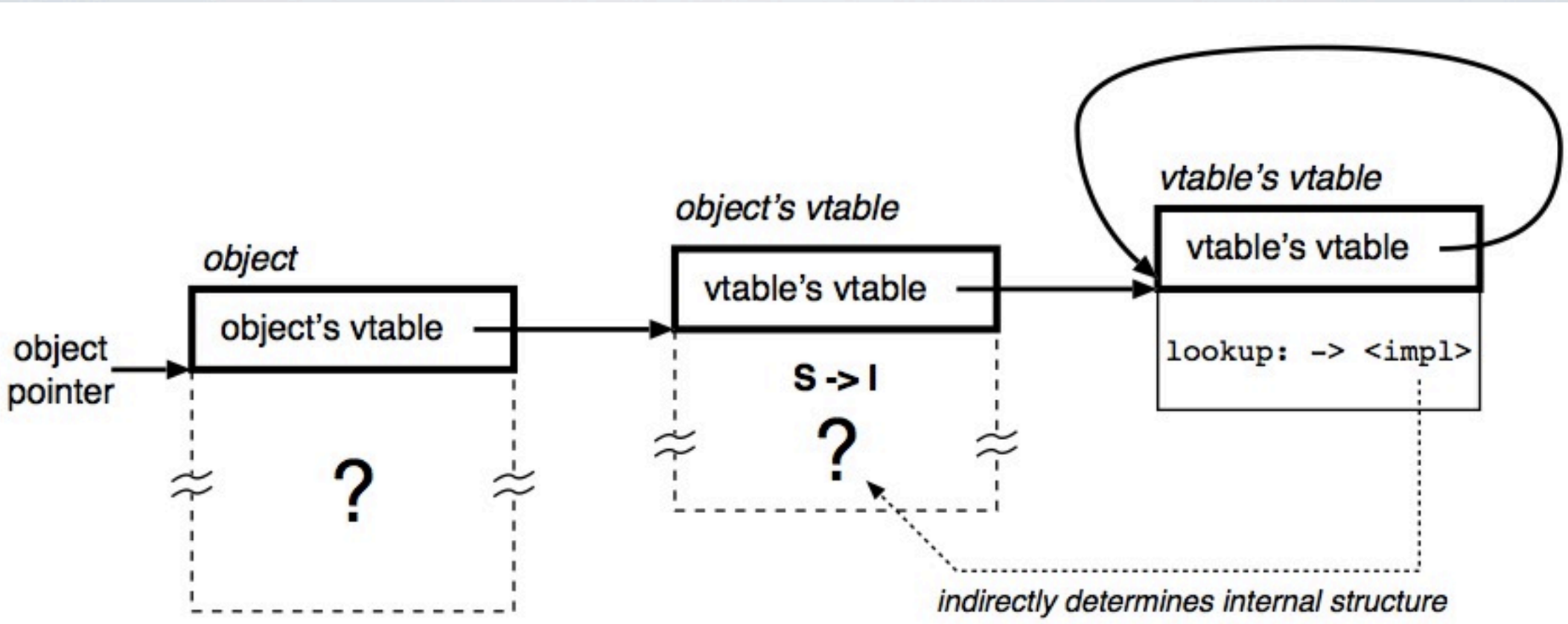
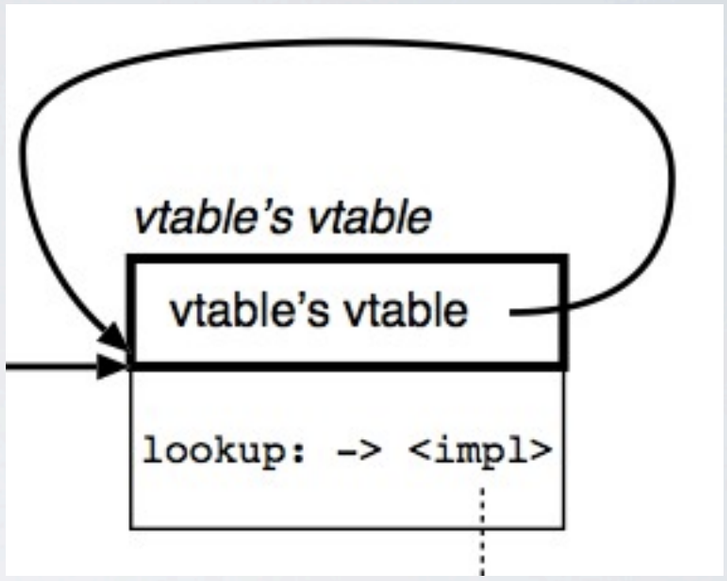


Figure 6. Everything is an object. Every object has a vtable that describes its behaviour. A method is looked up in a vtable by invoking its `lookup` method.

全てはオブジェクト.全てのオブジェクトにはその振る舞いを記述しているvtableがある.メソッドはそのルックアップメソッドを呼び出すことでvtableに見つけられる.

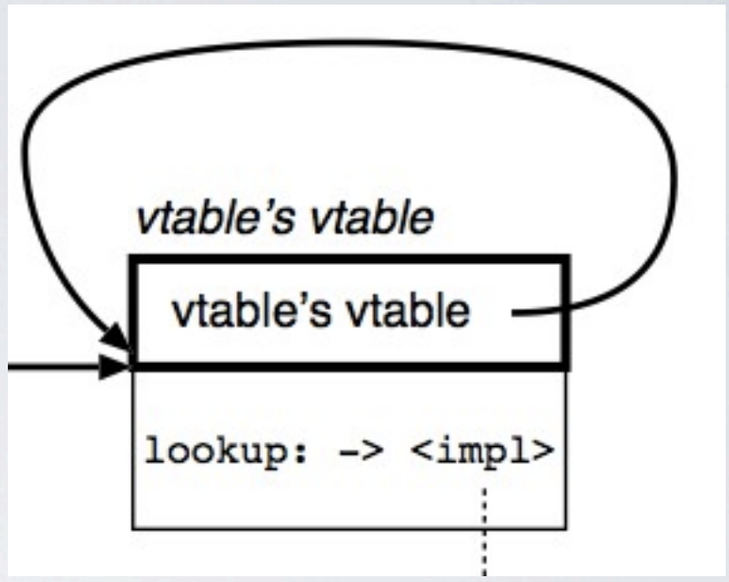
MOP



5 core metaclass methods 5つのコアメタクラスメソッド

- addMethod
- lookup
- allocate
- delegated
- intern

MOP



5 core metaclass methods 5つのコアメタクラスメソッド

- addMethod
- lookup
- allocate
- delegated
- intern

4 native (and jitted) VM ops 4つのネイティブ(とjitされた)

- SELF
- CLASS
- BIND
- MSG

meta is a method call, no native op

VM

- JIT default. For intel and powerpc. No arm yet.
  JIT デフォルト,intelと powerpc 用.arm はまだ

- Bytecode for unsupported CPUs, and for debugging
  サポートしていないCPUとデバッグ用に Bytecode

- Very simple. From lua ~50 ops. Do complicated stuff in the compiler (control) or library methods (array, hash, io, syscalls).
  とても簡単. lua から~50ops. 複雑なことは

```
/// PN_OP - a compressed three-address op (as 32bit int bitfield)
typedef struct {
  u8 code:8; ///< the op. See vm.c http://www.lua.org/doc/jucs05.pdf
  int a:12;  ///< the data (i.e the register)
  int b:12;  ///< optional arg, the message
} PN_OP;
```

DATA データ

- Primitive obj (in one word) vs extended objects (vt, uniq, size, data). プリミティブオブジェクト(1ワードで) vs 拡張されたオブジェクト
- INT, BOOL, NIL as primitives, everything else is an object.
  INT, BOOL, NILはプリミティブで,他の全てはオブジェクト
- last bits 00 => foreign ptr or our obj (in our memory pages)
  最後のビット 00 => 外のポインタか自分たちのオブジェクト(自分たちのメモリページ内の)
- last bits 10 => bool (true or false, unused by p2)

  最後のビット 10 => ブール (真か偽,p2では使われない)
- last bit 1 => int (shifted by 1) 最後のビット 1 => 整数 (1でシフトされた)
- Note: Different to dart, which has native int and shifts ptrs.
  ネイティブのintを持ち,ポインタをシフトするdartとは違う

010 could be used for short word-size strings. true then 110, false 0 == NIL

CALLING CONVENTION

呼出規約

only native, no stddecl, or foreign decl yet

- Native C cdecl (32bit) and fastcall (64bit) layout

  ネイティブのCのcdecl(32bit)とfastcall(64bit)のレイアウト

- Fast, and easy to interface, call-out and call-in.

  速く,インターフェースが簡単,call-out と call-in

  Fast function calls, no function call overhead (as in LISP)

  速い関数呼び出し,関数呼び出しのオーバーヘッドがない(LISPのように)

- OO: Every potion method prepends 2 args.

  OO: 全ての potion メソッドは 2つの引数を先頭に追加する

  interpreter, environment (a closure), self, optional args

  インタプリタ,環境(クロージャ),self,オプショナルな引数

only native, no stddecl, or foreign decl yet

GC - CHENEY LOOP

- walks the stack, not the heap, using volatile (not in regs)
  スタックを見る,ヒープではなく,揮発性のものを使う (レジスタではなく)

- copying (i.e. compacting), inprecise on ptrs, thread-friendly, fast (~**4ms** per GC) コピー(i.e.コンパクティング),ポインタで不正確,スレッドフレンドリー

- gc friendly data, chain of fwd ptr, gc フレンドリーなデータ,正方向ポインタの連鎖
  also for thread-shared data - parrot "proxy" スレッドで共有されたデータに対して

- i.e. essentially a tri-color algo 基本的には,tri-

- just not stop-the-world and mark&sweep, uses no private stack. data knows about threads, proxies

- no stop-the-world, mark&sweep, uses no private stack.
  data knows about threads, proxies. "mostly copying" or libgc look also good

GC

3 memory areas: 3つのメモリ領域

- protected segment (boot + core) 保護されたセグメント（ブート + コア）
  - birth segment (fast generation, minor collections) 生成セグメント（速い発生,マイナーコレクション）
  - main segment (major collections) メインセグメント（メジャーコレクション）

+ the other segment (mprotected) 古い: GC,mprotected中に生きているセグメン

- old: swapped out with live segments during GC, mprotected

DESIGN DECISIONS

設計の決定

- support 90% but do not sacrifice for the rest
  90%をサポートするが,残りのための犠牲にならない

- gmake and c99 gcc/clang/icc are everywhere
  gmake と c99 gcc/clang はどこでもある

- no MSVC, bsd make, no strict C++-only compilers
  MSVC,bsd make,厳密な C++ のみのコンパイラをサポートしない

- early testing with cross-compiling and threads
  クロスコンパイルとスレッドで早いテスト

not afterwards

FUNCTIONAL 機能

- use destruction with care.
  I use LISP names: nreverse, delete

- return copies, do not change arguments. コピーを返す.引数を変更しない

- Str immutable, Buf bytebuffers for io. 文字列 変更不可, バッファ io用バイトバッファ

- no functions. pass a message to everything. 関数なし.メッセージを全てに渡す

- no statements. everything is an expression. ステートメントなし.全ては式

- returns something and can be stacked

- use destruction with care. I use LISP names: nreverse, nsort, delete
  Str: fast cmp, threads friendly
  expr: returns something and can be stacked

MACROS マクロ

- With non-lisp languages: 非lisp言語で
- **parser macros** - *free* パーザーマクロ

  in parse context, use existing parser syntax: <rule> ...

  コンテキストをパーズするさいに,既存のパーザーシンタックス<rule>...を使う.

you can do everything: control constructs, like when, foreach, unless
start getting messy, where to be added into the parser state machine, fragile (messes with existing parser rules), and look bad because of the <rule> syntax.

- **compiler macros** - *limited* コンパイラマクロ

  like a function call. evaluate not all args, only some.

  関数呼び出しと似ている.全ての引数ではなく,いくつかのみを評価する

  body unquoting with `expr`. ボディは`expr` で引用を終える

2nd type: call list block - control constructs

limited to calls. but if your parser does nothing else calls (like lisp does), its the perfect point to add it.
do not change the parser statemachine, just hook in the compiler.

you can do everything: control constructs, like while, foreach, unless.
starts getting messy. where to be added into the parser state machine, fragile (messes with existing parser rules), and look bad because of the <rule> syntax. needs parser support, not pre-compiled.

limited to calls. but if your parser does nothing else then calls (like lisp does), its the perfect point to add it.
do not change the parser statemachine, just hook into the compiler.

MACROS

```
$a = if (0) { 12 }
elsif (1) { 14 }
else { 16 }
```

```
ifstmt = IF e:ifexpr s:block - !"els"  { $$ = PN_OP(AST_AND, e, s) }
   | IF e:ifexpr s1:block -
       { $$ = e = PN_AST3(MSG, PN_if, PN_AST(LIST, PN_TUP(e)), s1) }
     (ELSIF e1:ifexpr f:block -
       { $$ = e = PN_PUSH(PN_TUPIF(e), PN_AST3(MSG, PN_elsif, PN_AST(LIST, PN_TUP(e1)), f)) } )*
     (ELSE s2:block
       { $$ = PN_PUSH(PN_TUPIF(e), PN_AST3(MSG, PN_else, PN_NIL, s2)) } )?

ifexpr = '(' - expr - ')' -
```

```
(assign (msg ("$a")
  expr (msg ("if" list (expr (value (0))) block (expr (value (12))))),
  msg ("elsif" list (expr (value (1))) block (expr (value (14))))),
  msg ("else" undef block (expr (value (16))))))
```

MACROS

```
$a = if ($DEBUG) { call(debug) }
else { callfast() }


macro ifdebug(ifblock, elseblock) {
  if ($DEBUG) { `ifblock` }
  else { `elseblock` }
}
```

`if ($DEBUG)..`- compile-time

`` `ifblock` `` - exec at run-time

MACROS

```
$a = if ($DEBUG) { call(debug) }
else { callfast() }

(assign (msg ("$a")
  expr (msg ("if" list (expr (msg ("$DEBUG")))
                  block (expr (msg ("call" list (expr (msg ("debug"))) undef)))),
  msg ("else" undef
                  block (expr (msg ("callfast" list undef undef)))))))


macro ifdebug(ifblock, elseblock) {
  if ($DEBUG) { `ifblock` }
  else { `elseblock` }
}

ifdebug( call(debug),
         callfast() ):
```

STATUS ステータス

STATUS ステータス

- potion maintenance moved to perl11.org
- greg upstream commits: better error handling, diagnostics
  greg アップストリームコミット：よりよいエラーハンドリング,診断
- release potion 0.1 next month potion 0.1 を来月にリリース

- more potion examples and features: より多くのpotionの例と機能
  ffi, debugger, coro/threads, UI bindings, shootout samples

TODO

- GOAL: parse 50% of p5 by Summer 2013 ✓, 90% by 2014.
  2013年の夏までにp5の50%をパーズ✓ 2014までに90%
- Parser:
- Compiler:
- VM:
- Libs:

TODO

- GOAL: parse 50% of p5 by Summer 2013 ✓, 90% by 2014.
- Parser: 50%. dynamic namespaces, refs, minor stuff
  パーザー：50%.動的な名前空間,リファレンス,マイナーなもの
- Compiler:
- VM:
- Libs:

TODO

- GOAL: parse 50% of p5 by Summer 2013 ✓, 90% by 2014.
- Parser: 50%. dynamic namespaces, refs, minor stuff
- Compiler: bytecode serialization, vm and jit✓
  Todo: constant folding, dead-code, macros, emit C or native.
  コンパイラ：バイトコードシリアライゼーションのみ.vmとjit.
  Todo：定数折畳,デッドコード,マクロ,Cかネイティブコードの実行
- VM:
- Libs:

TODO

- GOAL: parse 50% of p5 by Summer 2013 ✓, 90% by 2014.
- Parser: 50%. dynamic namespaces, refs, minor stuff
- Compiler: bytecode serialization, vm and jit✓
  Todo: constant folding, dead-code, macros, emit C or native.
- VM: arm jit, threads, stabilize callcc, ffi (30%), debugger (50%)
  VM: arm jit, スレッド,安定した callcc, ffi, デバッガ
- Libs:

TODO

- GOAL: parse 50% of p5 by Summer 2013 ✓, 90% by 2014.
- Parser: 50%. dynamic namespaces, refs, minor stuff
- Compiler: bytecode serialization, vm and jit✓
  Todo: constant folding, dead-code, macros, emit C or native.
- VM: arm jit, threads, stabilize callcc, ffi (30%), debugger (50%).
- Libs: aio ✓, buffile ✓, sprintf (20%), pcre (10%), bignum (20%), p5 compat. p5 との互換性

QUESTIONS

perl11.org/p2/