S3, Tokyo, 2010 – 09 – 27

# PEG-based transformer provides front-, middle- and back-end stages in a simple compiler

Ian Piumarta
Viewpoints Research Institute
ian@vpri.org

# why design your own programming language?


i.e., *syntax matters*

# why design your own programming language?

mathematica

```
f = x Sin[b x]
Plot[ D[f, x], 0, 10 ]
```

APL

```
N ← 4 5 6 7
     ⇒ 4 5 6 7

N + 4
     ⇒ 8 9 10 11

+/N
     ⇒ 22
```

*but...*

# apl

```
X[⍋X+.≠' ';]
```

```
(~R∊R∘.×R)/R←1↓⍳R
```

# apl

*and...*


*!!!*

# mainstream

programming

- `lex`, `yacc`
- `sed`, `awk`
- `cpp`, `m4`
- `make`
- `autoconf`, `automake`, `CMake`
- `sh`, `ksh`, `bash`

or, less obviously. . .

# levels of abstraction and specific representations

## basic instruction formats of the PowerPC

```
#define _I(  OP,          BD,AA,LK) _GEN((_u6(OP)<<26)|                                                     _d26(BD)|    (_u1(AA)<<1)|_u1(LK))
#define _B(  OP,BO,BI,    BD,AA,LK) _GEN((_u6(OP)<<26)|(_u5(BO)<<21)|(_u5(BI)<<16)|                         _d16(BD)|    (_u1(AA)<<1)|_u1(LK))
#define _D(  OP,RD,RA,          DD) _GEN((_u6(OP)<<26)|(_u5(RD)<<21)|(_u5(RA)<<16)|                         _s16(DD)                          )
#define _Du( OP,RD,RA,          DD) _GEN((_u6(OP)<<26)|(_u5(RD)<<21)|(_u5(RA)<<16)|                         _u16(DD)                          )
#define _Ds( OP,RD,RA,          DD) _GEN((_u6(OP)<<26)|(_u5(RD)<<21)|(_u5(RA)<<16)|                        _su16(DD)                          )
#define _X(  OP,RD,RA,RB,    XO,RC) _GEN((_u6(OP)<<26)|(_u5(RD)<<21)|(_u5(RA)<<16)|( _u5(RB)<<11)|                  (_u10(XO)<<1)|_u1(RC))
#define _XL( OP,BO,BI,       XO,LK) _GEN((_u6(OP)<<26)|(_u5(BO)<<21)|(_u5(BI)<<16)|( _u5(00)<<11)|                  (_u10(XO)<<1)|_u1(LK))
#define _XFX(OP,RD,          SR,XO) _GEN((_u6(OP)<<26)|(_u5(RD)<<21)|              (_u10(SR)<<11)|                  (_u10(XO)<<1)|_u1(00))
#define _XO( OP,RD,RA,RB,OE,XO,RC) _GEN((_u6(OP)<<26)|(_u5(RD)<<21)|(_u5(RA)<<16)|( _u5(RB)<<11)|(_u1(OE)<<10)|( _u9(XO)<<1)|_u1(RC))
#define _M(  OP,RS,RA,SH,MB,ME,RC) _GEN((_u6(OP)<<26)|(_u5(RS)<<21)|(_u5(RA)<<16)|( _u5(SH)<<11)|(_u5(MB)<< 6)|( _u5(ME)<<1)|_u1(RC))
#define _VX( OP,VD,VA,VB,       XO) _GEN((_u6(OP)<<26)|(_u5(VD)<<21)|(_u5(VA)<<16)|( _u5(VB)<<11)|                          _u11(XO))
```

## and their use as a DSL for describing opcodes

```
#define ADDrrr(RD, RA, RB)         _XO  (31, RD, RA, RB, 0, 266, 0)
#define ADDIrri(RD, RA, IMM)       _D   (14, RD, RA, IMM)
#define ADDISrri(RD, RA, IMM)      _Ds  (15, RD, RA, IMM)
#define ANDrrr(RA, RS, RB)         _X   (31, RS, RA, RB,  28, 0)
#define ANDI_rri(RA, RS, IMM)      _Du  (28, RS, RA, IMM)
#define Bi(BD)                     _I   (18, BD, 0, 0)
#define BCLRii(BO,BI)              _XL  (19, BO, BI,  16, 0)
#define CMPiirr(CR, LL, RA, RB)    _X   (31, ((CR)<<2)|(LL), RA, RB, 0, 0)
#define CMPIiiri(CR, LL, RA, IMM)  _D   (11, ((CR)<<2)|(LL), RA, IMM)
 ...
#define TWirr(TO,RA,RB)            _X   (31, TO, RA, RB,   4, 0)
#define TWIiri(TO,RA,IM)           _D   (03, TO, RA, IM)
#define XORrrr(RA,RS,RB)           _X   (31, RS, RA, RB, 316, 0)
#define XORIrri(RA,RS,IM)          _Du  (26, RS, RA, IM)
```

# cpp macros to generate code, generated from...

```
#include "asm-i386.h" /* #cpu pentium */

#include <stdio.h>

static char code[1024];

typedef void (*pvf)(void);               /* Pointer to Void Function */

int main()
{
  pvf myFunction= (pvf)code;             /* the generated function */
  void *loop;                            /* labels */
    _ASM_APP_1
  _ASM_ORG(myFunction);
        PUSHLr  (_EBP);
        MOVLrr  (_ESP, _EBP);
        PUSHLr  (_EBX);
        MOVLir  ('a', _EBX);
  _ASM_DEF(loop);       PUSHLr  (_EBX);
        CALLm   (putchar,0,0,0);
        ADDLir  (1, _EBX);
        CMPLir  ('z'+1, _EBX);
        JNEm    (loop,0,0,0);
        PUSHLi  (10);
        CALLm   (putchar,0,0,0);
        POPLr   (_EBX);
        LEAVE   ();
        RET     ();
  _ASM_NOAPP_1
  myFunction();
  return 0;
}
```

# C program with i386 “DSL”

```
#cpu pentium

#include <stdio.h>

static char code[1024];

typedef void (*pvf)(void);                   /* Pointer to Void Function */

int main()
{
  pvf myFunction= (pvf)code;                 /* the generated function */
  void *loop;                                /* labels */
  #[
        .org    myFunction                   # generate code at this address
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        movl    $'a', %ebx
  loop: pushl   %ebx
        call    putchar
        addl    $1, %ebx
        cmpl    $'z'+1, %ebx
        jne     loop
        pushl   $10
        call    putchar
        popl    %ebx
        leave
        ret
  ]#
  myFunction();
  return 0;
}
```

# goal of today’s presentation

be (half?) convinced that “rolling your own” language isn’t hard

look at the one I made

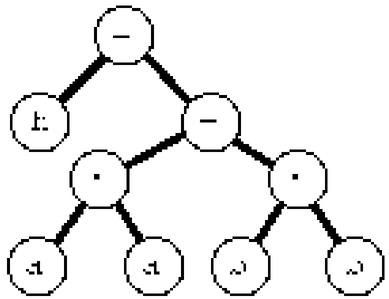
go home and build your own better one!

# specific program abstractions & representations

source text

```
h := a * a + b * b ;
```

syntax tree



abstract machine

```
push b; push b; mul; add
```

concrete machine

```
movl b, 4(%esp)
movl b, %eax
mull 4(%esp), %eax
addl 0(%esp), %eax
```

# single flexible representation

source

- files or string

⇒ sequences of characters in input program

if

sequence = `( ... )`
character = $'x'$

then the program

```
let x = 42;
```

is represented by the sequence

`( 'l' 'e' 't' '␣' 'x' '␣' '=' '␣' '4' '2' ';' )`

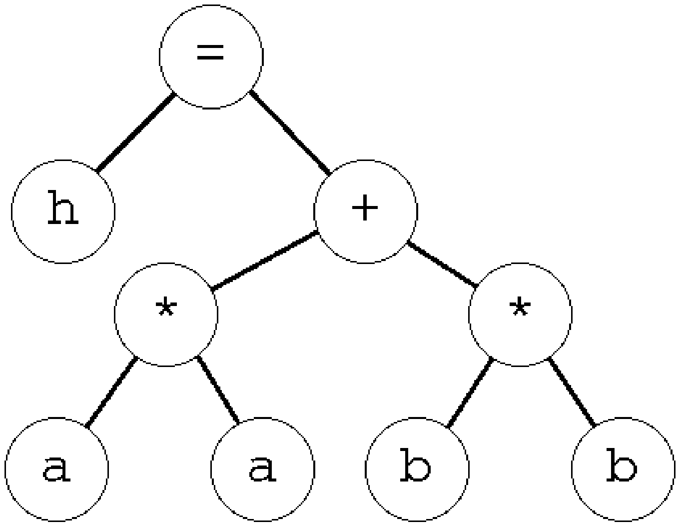
which (when unambiguous) we will write

**(** *l e t ␣ x ␣ = ␣ 4 2 ;* **)**

## single flexible representation

ASTs are sequences of names, literals and sub-ASTs

```
h = a * a + b * b ;
```




```
(set h
     (+ (* a a)
        (* b b)))
```

# single flexible representation

abstract machine

- sequence of symbolic instructions and operands

```
( push a push a mul
  push b push b mul
  add
  store h )
```

## single flexible representation

concrete machine

- sequences of characters in output assembly language
- file or string

```
( m o v l ␣ b , % e a x
  a d d l ␣ 4 ( % e a x ) , % e a x )
```

⇒

```
movl b,%eax
addl 4(%eax),%eax
```

## single flexible representation

at each stage

- input is a list of objects
- output is a list of objects

each stage is a transformation from lists to lists

- recognise an input list structure
- generate a related output list strusture

# parsing expressions

similar to regular expressions

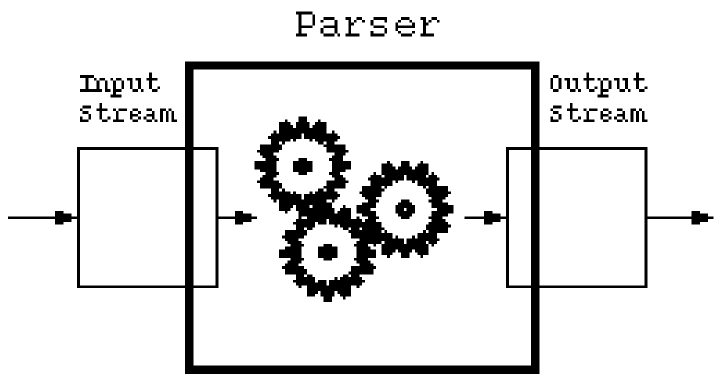
expressions can be *named*, like grammar “rules”

recursive-descent parsers are described *trivially* and *directly*

extensions: ours will describe *input patterns* **and** *output templates*

```
rule = input-pattern-1 -> output-template-1
     | input-pattern-2 -> output-template-2
     | input-pattern-3 -> output-template-3
    ...
```

# parsers operate on streams of objects



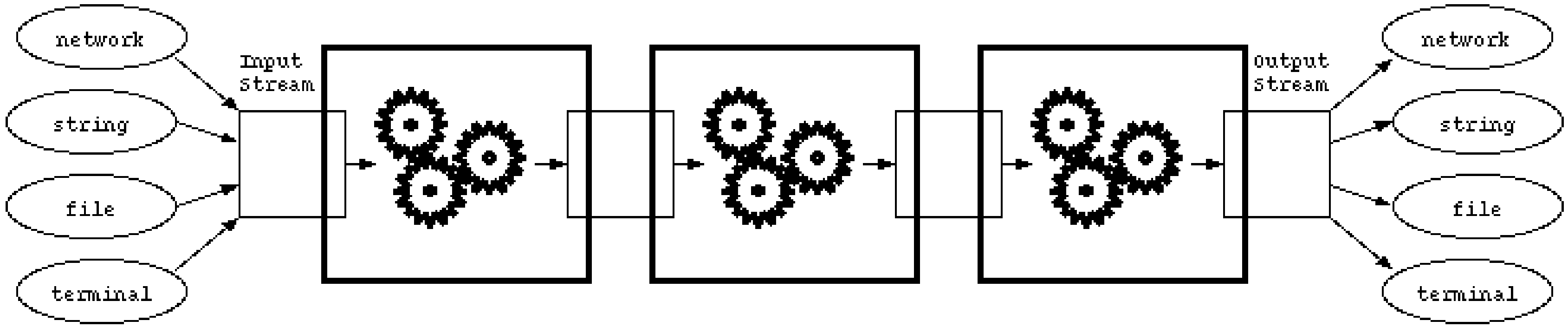

objects subsume character sets

- trivial to support full Unicode: non-latin character sets

# parsers can be assembled into pipelines

share a stream between two parsers:

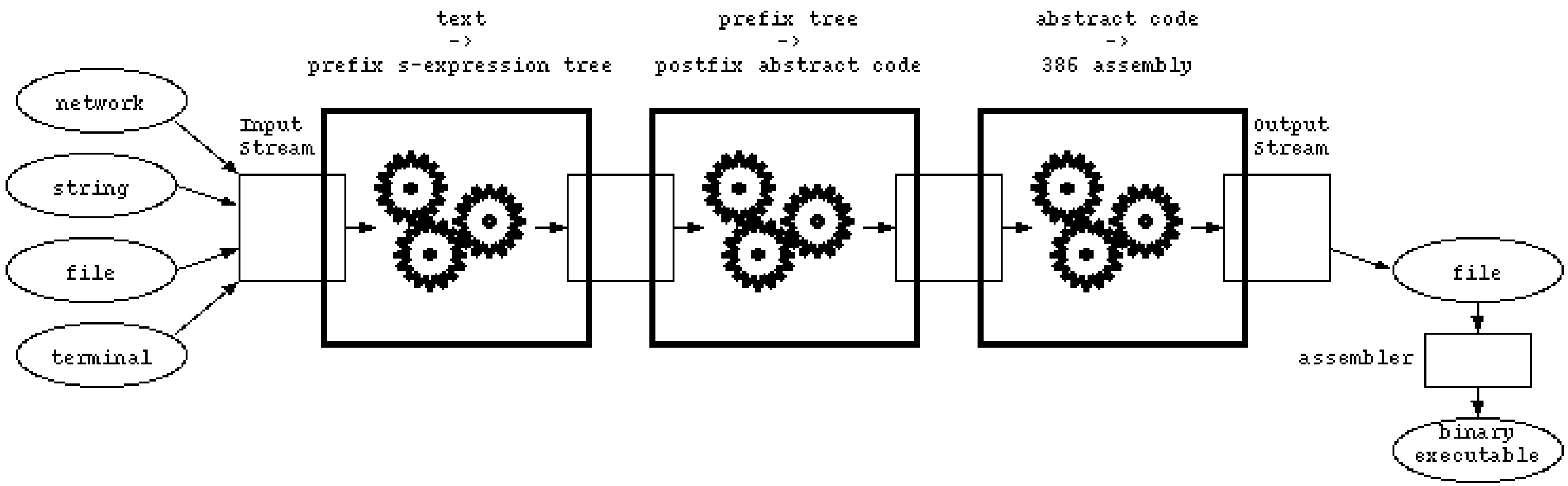
- first parser's output stream is
- next parser's input stream



# each parser transforms a DSL into another DSL

each stage in the pipeline

- recognises a *specific representation* of the semantics
- generates a different *specific representation* of the semantics




level of abstraction decreases towards the right

# PEG-based pattern matching

a *parser* is described by a collection of named *rules*

| | |
|---|---|
| match anything | `.` |
| match a character | `"c"`<br>`[0123456789abcdef]`<br>`[0-9a-f]` |
| match a string | `"a string"` |
| match a named expression | *rule-name* |
| match zero or one | *expression* `?` |
| match zero or more | *expression* `*` |
| match one or more | *expression* `+` |
| predicate “not” | `!` *expression* |
| match two expressions (“and”) | *expr1 expr2* |
| match one of two alternatives (“or”) | *expr1* `\|` *expr2* |
| grouping | `(` *expr* `...` `)` |
| naming an expression | *rule-name* `=` *expression* |

# example input program

```
nfibs = function(n)
  if n < 2
  then 1
  else
    nfibs(n - 2) + nfibs(n - 1) + 1;

print nfibs(32)
```

or the equivalent

```
(define nfibs
  (lambda (n)
    (if (< n 2)
      1
      (+ 1 (+ (nfibs (- n 1)) (nfibs (- n 2)))))))

(print (nfibs 32))
```

which is (slightly) easier to parse and will serve as our example

# recognise text description of a parse tree

```
start    = sexpr

sexpr    = spacing (list | atom)

list     = "(" sexpr* spacing ")"

atom     = symbol | number

symbol   = letter (letter | digit)*

number   = digit+

letter   = [-+!$%&*./:<=>?@A-Z\\^_a-z|~]
digit    = [0-9]

spacing  = [ \t\n\r]*
```

```
(define nfibs
  (lambda (n)
    (if (< n 2)
      1
      (+ 1
        (+ (nfibs (- n 1))
           (nfibs (- n 2))
  ))))))
```

# input pattern + output template = *structure transformation*

every expression generates a *result*

- results are *objects* or *sequences* of objects
- results can be stored in *variables*
- results can be transformed in various ways

| | |
|---|---|
| make the result be the matched sequence | `expression $` |
| make the result be the matched symbol | `expression $$` |
| convert result sequence to an integer, base 10 | `#10` |
| save the current result in a variable | `:variable-name` |
| load a new current result from a variable | `-> :variable-name` |

the result of repetition (`?`, `+` and `*`) is a *sequence* of the results of the iterated expression

the result of a named expression is the result of its last sub-expression

# transform text into parse tree

recogniser as shown earlier + “result” operators (in **bold**)

```
start    = sexpr

sexpr    = spacing (list | atom)

list     = "(" sexpr* :l spacing ")" -> :l

atom     = symbol | number

symbol   = ( letter (letter | digit)* ) $$

number   = digit+ $ #10

letter   = [-+!$%&*./:<=>?@A-Z\\^_a-z|~]
digit    = [0-9]

spacing  = [ \t\n\r]*
```

# example program's parse tree

parse tree printable representation looks just like source program

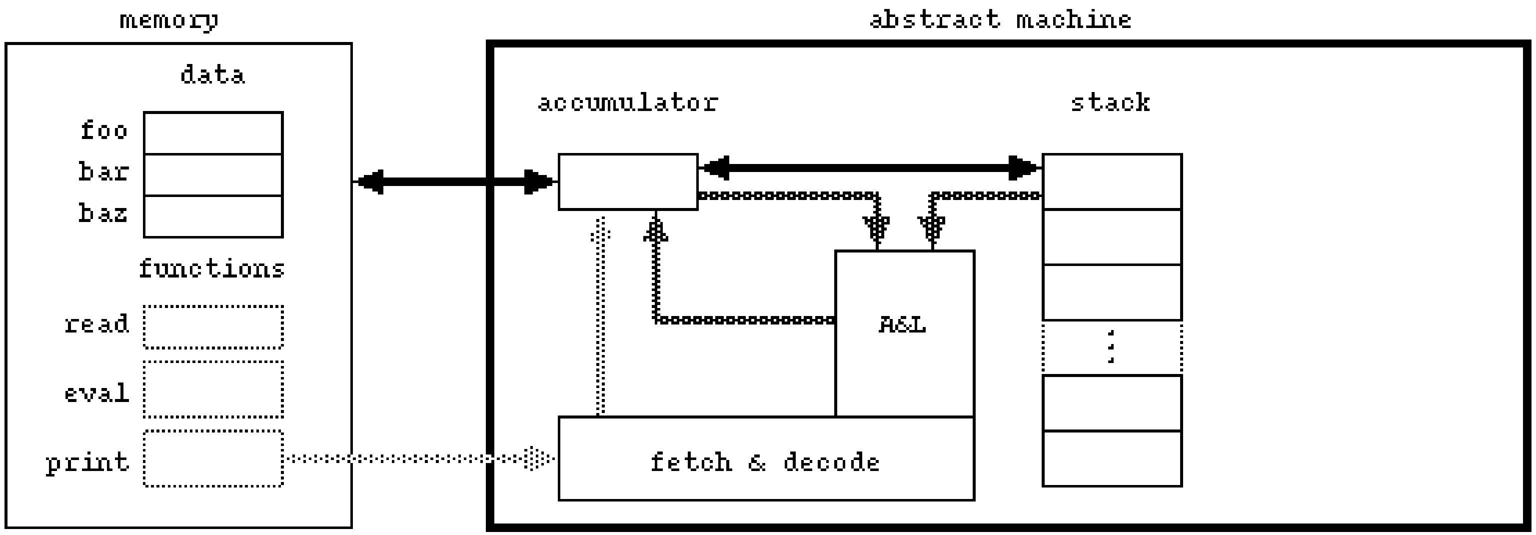
```
(define nfibs
        (lambda (n)
                (if (< n 2)
                    1
                    (+ 1
                       (+ (nfibs (- n
                                    1))
                          (nfibs (- n
                                    2))))))))

(print (nfibs 32))
```

next: invent abstract target "machine" to execute this program

# abstract machine

abstract machine model is close to semantics of source langauge




- single-instruction function call, prologue, and epilogue
- single-instruction load/store of argument, temporary, global variables
- instructions operate on accumulator and stack

# abstract machine accumulator and stack

accumulator holds “current” result

- result of fuction calls
- output from operators
- first input to operators

stack for intermediate results

- push accumulator
- second input to binary operators

stack for function activations

- function call with arguments on stack
- function prologue (enter), epilogue (return)

# abstract machine instructions

| | |
|---|---|
| **`load-long 32`** | load literal into accumulator |
| **`load-var print`** | load global variable |
| `load-arg 0` | load argument |
| **`save`** | push accumulator onto stack |
| `add` | pop and add to accumulator |
| `sub` | pop and subtract from accumulator |
| `less` | pop, compare ’<’, and set accumulator 0 or 1 |
| **`call 1`** | call accumulator with N arguments |
| `enter` | function prologue |
| `arg-name x` | “declare” argument name |
| `leave` | function epilogue (return) |
| `label 3` | define a label |
| `branch 2` | branch to a label |
| `branch-false 1` | branch if accumulator 0 |
| `load-label 3` | load address into accumulator |
| `long nfibs` | declare a global variable |
| `store-var nfibs` | store accumulator into variable |
| `main` | begin main program |
| **`exit`** | exit from main program |

# transformation to abstract machine code

*parse tree*

```
(define nfibs
  (lambda (n)
    (if (< n 2)
      1
      (+ (+ (nfibs (- n 1))
            (nfibs (- n 2)))
         1)))))

(print (nfibs 32))
```

*abstract machine code*

```
(label 3
 enter
 load-long 2 save load-arg 0 less
   branch-false 1
   load-long 1 branch 2
   label 1
 load-long 1 save load-arg 0 sub save
   load-var nfibs call 1 save
 load-long 2 save load-arg 0 sub save
   load-var nfibs call 1 add save
 load-long 1 add
 label 2
 leave

 main

 long nfibs load-label 3 store-var nfibs

 load-long 32 save
   load-var nfibs call 1 save
   load-var print call 1
 exit)
```

# more parsing expressions and output templates

additional parsing expressions

- structure matching
- structure generating
- predicates

| | |
|---|---|
| predicate “lookahead for ...” | `&` *`expression`* |
| execute host language expression | `` ` ``*`expression`* |
| predicate “is a symbol” | ``&`symbol?`` |
| execute/generate arbitrary value | ``->`(list-length x)`` |
| match structure | `'(` *`expressions ...`* `)` |
| generate literal structure | `-> ()` |
| | `-> ( a b c )` |
| (with variable substitution) | `-> ( a :b c )` |
| (flattened once) | `-> ( a ::b c )` |
| (flattened twice) | `-> ( a :::b c )` |

# flattening generated structure

flattening structure

- let `b` contain `((x y z))`

```
-> ( a :b c )       result is   (a ((x y z)) c)
-> ( a ::b c )                  (a (x y z) c)
-> ( a :::b c )                 (a x y z c)
```

# transform tree into abstract machine

```
start = expr

expr = long:x                         -> (load-long :x)
  |    name:x &`(is-arg x):n          -> (load-arg  :n)
  |    name:x                         -> (load-var  :x)
  | '( '< expr:x expr:y )             -> (::y save ::x less)
  | '( '+ expr:x expr:y )             -> (::y save ::x add)
  | '( '- expr:x expr:y )             -> (::y save ::x sub)
  | '( 'define name:n expr:e )        -> (long :n ::e store-var :n)

  | '( 'lambda '(params) expr*:b )  -> (enter :::b leave):l
                                      ->`(save-lambda l):n
                                      -> (load-label :n)

  | '( 'if expr:t expr:x expr:y )   ->`(new-label):a
                                      ->`(new-label):b
                                      -> (          ::t branch-false :a
                                                    ::x branch :b
                                               label :a ::y
                                               label :b )

  | '( expr:f &arity:n args:a )     -> (::a ::f call :n)
```

# transform tree into abstract machine

```
long     = &`long?    .
name     = &`symbol? .

arity    = .*:x                              ->`(list-length x)

args     = expr:e args:a                     -> (::a ::e save)
         | expr:e                            -> (::e save)
         |                                   -> ()

params   = ( name:h params:t | name:h ) ->`(arg-name h)
         |
```

# function call transformation

| *parse tree (inverted)* | *input matches* | *generated output* |
| --- | --- | --- |
| `2)))` | `long` | `load-long 2` |
| `n` | `name is-arg` | `save load-arg 0` |
| `(-` | `'- args` | `sub save` |
| `(nfibs` | `name` *`(!is-arg)`* | `load-var nfibs` |
| | `'(expr arity` | `call 1` |

relevant parts of the transformation grammar:

```
expr = long:x                               -> (load-long :x)
  |    name:x &`(is-arg x):n                -> (load-arg  :n)
  |    name:x                               -> (load-var  :x)
  | '( '- expr:x expr:y )                   -> (::y save ::x sub)
  | '( expr:f &arity:n args:a )             -> (::a ::f call :n)

arity   = .*:x                              ->`(list-length x)

args    = expr:e args:a                     -> (::a ::e save)
        | expr:e                            -> (::e save)
        |                                   -> ()
```

# transformation to abstract machine code

*parse tree* | *abstract machine code*

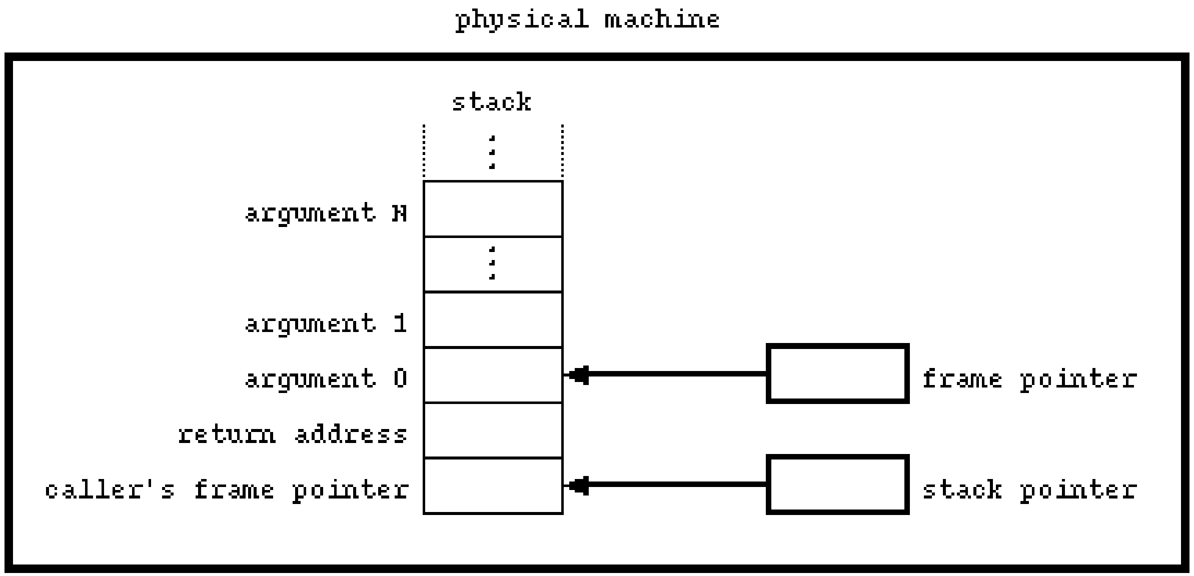
```
(define nfibs
                                (label 3
  (lambda (n)                    enter
    (if (< n 2)                  load-long 2 save load-arg 0 less
                                   branch-false 1
      1                            load-long 1 branch 2
                                   label 1
      (+ (+ (nfibs (- n 1))      load-long 1 save load-arg 0 sub save
                                   load-var nfibs call 1 save
            (nfibs (- n 2)))     load-long 2 save load-arg 0 sub save
                                   load-var nfibs call 1 add save
         1)))))                  load-long 1 add
                                 label 2
                                 leave

                                 main

                                 long nfibs load-label 3 store-var nfibs

(print (nfibs 32))               load-long 32 save
                                   load-var nfibs call 1 save
                                   load-var print call 1
                                 exit)
```

# target machine model




physical registers assigned to abstract registers:

| | |
|---|---|
| accumulator | `eax` |
| stack pointer | `ebx` |
| frame pointer | `esi` |

# transformation to i386 machine code

```
(print (nfibs 32))
```

*parse tree (inverted)* — *abstract code* — *i386 machine code*

```
      32))             ( load-long 32        movl    $32, %eax
                         save                subl    $4, %ebx
                                             movl    %eax, (%ebx)
    nfibs                load-var nfibs      movl    nfibs, %eax
  (                      call 1              call    *%eax
                                             addl    $4, %ebx
                         save                subl    $4, %ebx
                                             movl    %eax, (%ebx)
 print                   load-var print      movl    print, %eax
(                        call 1              call    *%eax
                       )                     addl    $4, %ebx
```

# string generator expressions

generation of string (sequence of characters) on output

| *value* | *generator* | *output* |
|---|---|---|
| literal string | `` `"abcdef" `` | `abcdef` |

with values substiuted from variables

| *variable value* | *generator* | *output* |
|---|---|---|
| (string) s = `"xyz"` | `` `"abc\${s}def" `` | `abcxyzdef` |
| (number) n = `42` | `` `"abc\#{n}def" `` | `abc42def` |

# transformation to i386 machine code

```
start =  insn*

insn  = 'load-long  .:l              `" movl $#l, %eax"
      | 'load-var   .:n              `" movl $n, %eax"
      | 'save                        `" subl $4, %ebx"
                                     `" movl %eax, (%ebx)"
      | 'call    .:n ->`(* 4 n):n `" call *%eax"
                                     `" addl $#n, %ebx"
      | ...
```

# transformation to i386 machine code

| *abstract code* | *i386 machine code* |
|---|---|
| ( load-long 32 | movl $32, %eax |
| save | subl $4, %ebx |
| | movl %eax, (%ebx) |
| load-var nfibs | movl nfibs, %eax |
| call 1 | call *%eax |
| | addl $4, %ebx |
| save | subl $4, %ebx |
| | movl %eax, (%ebx) |
| load-var print | movl print, %eax |
| call 1 | call *%eax |
| ) | addl $4, %ebx |

relevant parts of the transformation grammar:

```
insn  = 'load-long  .:l              `" movl $#l, %eax"
      | 'load-var   .:n              `" movl $n, %eax"
      | 'save                        `" subl $4, %ebx"
                                     `" movl %eax, (%ebx)"
      | 'call   .:n ->`(* 4 n):n `" call *%eax"
                                     `" addl $#n, %ebx"
      | ...
```

## summary

from source to assembly language

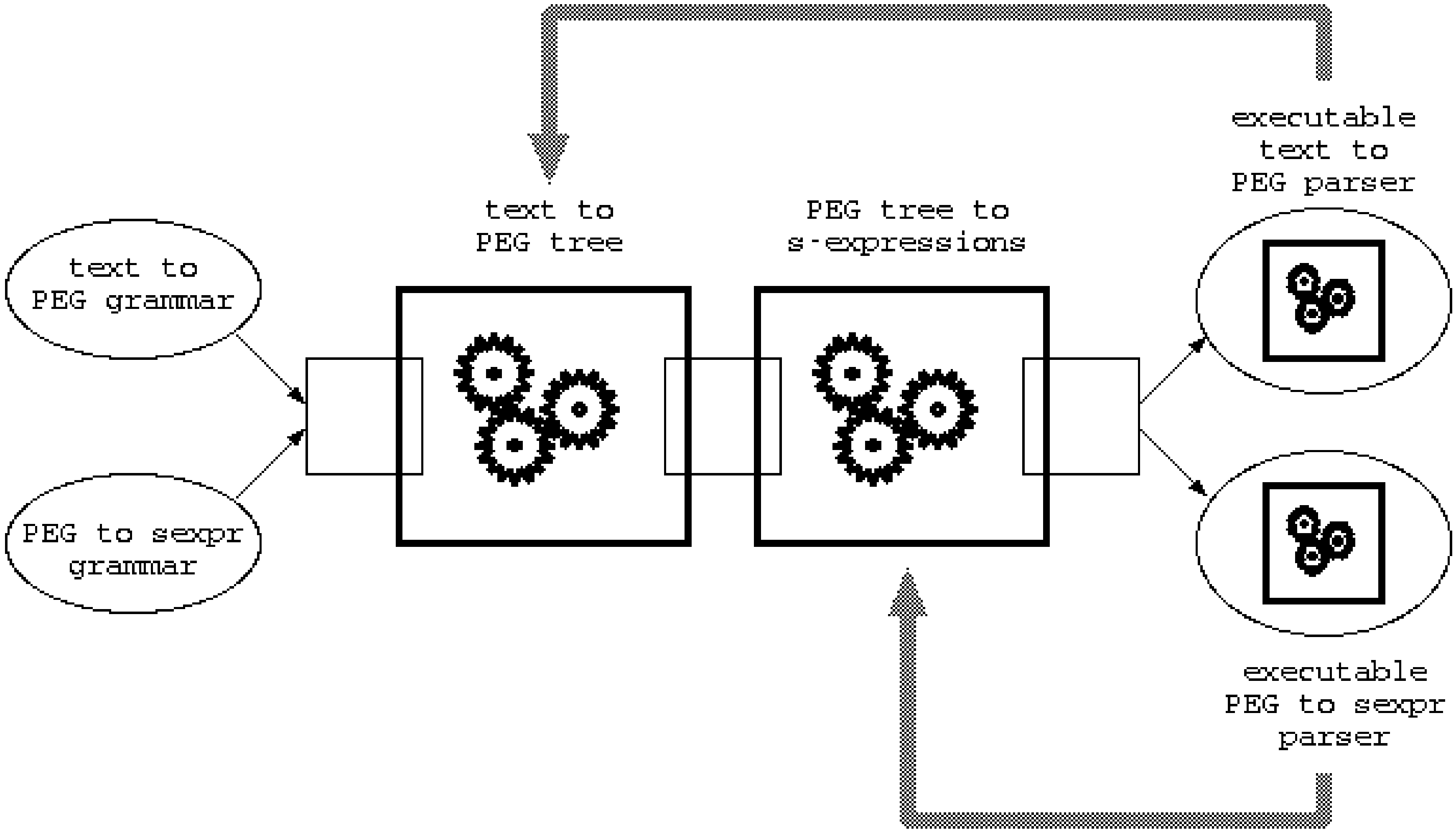
- tree to abstract code
  - = 30 lines of matcher-generator
  - + 1 line per additional arithmetic operator

- abstract to i386 code
  - = 60 lines of matcher-generator
  - + 2 lines per additional arithmetic operator

illustration with general-purpose language

- might be similar or (much) longer
- depending on semantics and supporting infrastructure

# parser/generator is built using itself



# limitations

optimisations

- peephole by additional pass
- bottom-up rewrite possible, but

  severe backtracking

  $\Rightarrow$ aggressive optimisation/memoisation in parser implemenation

dynamic types: limitations depend on architectural choices; e.g.

- conservative GC $\Rightarrow$ no complications
- precise GC $\Rightarrow$ compiler produces stack maps, etc.

static types

- simple type system (synthesis + tag) “doable”
- powerful type system requires more than pattern matching
  - combination/reinterpretation of operator/argument attributes
  - closer to dissimilar evaluation in multiple namespaces

# to do: PEGs as finite state machines

regular expressions can be *very* fast

- convert RegExp into non-deterministic FSM
- incrementally convert non-deterministic FSM into deterministic FSM at run time
- recognition of input in $O(n)$ time

simple, incremental algorithm

- how can the *ordered choice* and *backtracking* of PEGs be incorporated into this model without loss of performance?

biggest problem: submatch tracking to generate '$' results

- tagged-transition NFAs

PEGs should be competitive with table-driven LR parsers.

# download and play with it

`http://piumarta.com/S3-2010`