How to Eat
a
Punie Elephant

Allison Randal
The Perl Foundation &
O'Reilly Media, Inc.

Perl 1

- December, 18th 1987
- Patches 1-14
- Baby Perl

Perl 1

- Variables

```
$scalar
@array
$array[0]
%hash
$hash{'key'}
```

Perl 1

- Operators

```
+ - * / % ++ -- x . & | ^ = += -= *= /= %=
&= |= ^= .= .. << >> ! ~ == != =~ !~ > < >=
<= eq ne gt lt ge le && || ?:
```

- Builtins

```
shift push pop split join chop
open close seek tell stat eof
```

Perl 1

- Patterns

```
$_ = 'test';
if (/^test/) { print "ok 1\n"; }

$a =~ s/a/x/g;
```

Perl 1

- Formats

```
format one =
@<<<
$foo
.

...
$~ = 'one';
write;
```

Perl 1

- Conditionals

```
if ($x == $y) {
    ...
} elsif ($x == $z) {
    ...
} else {
    ...
}

unless ($x == $y) { ... }
```

Perl 1

- Loops

```
while ($test = shift) {
    print “$test...”;
    ...
}

until ($x == 42) {
    ...
    $x++;
}
```

Perl 1

- Loops

```
for ($i = 0; $i < 10; $i++) {
    ...
}

for (@array) {
    ...
}
```

Perl 1

- Subroutines

```
sub foo {
    print $_[0];
}

$result = do foo($x);
```

Perl 1

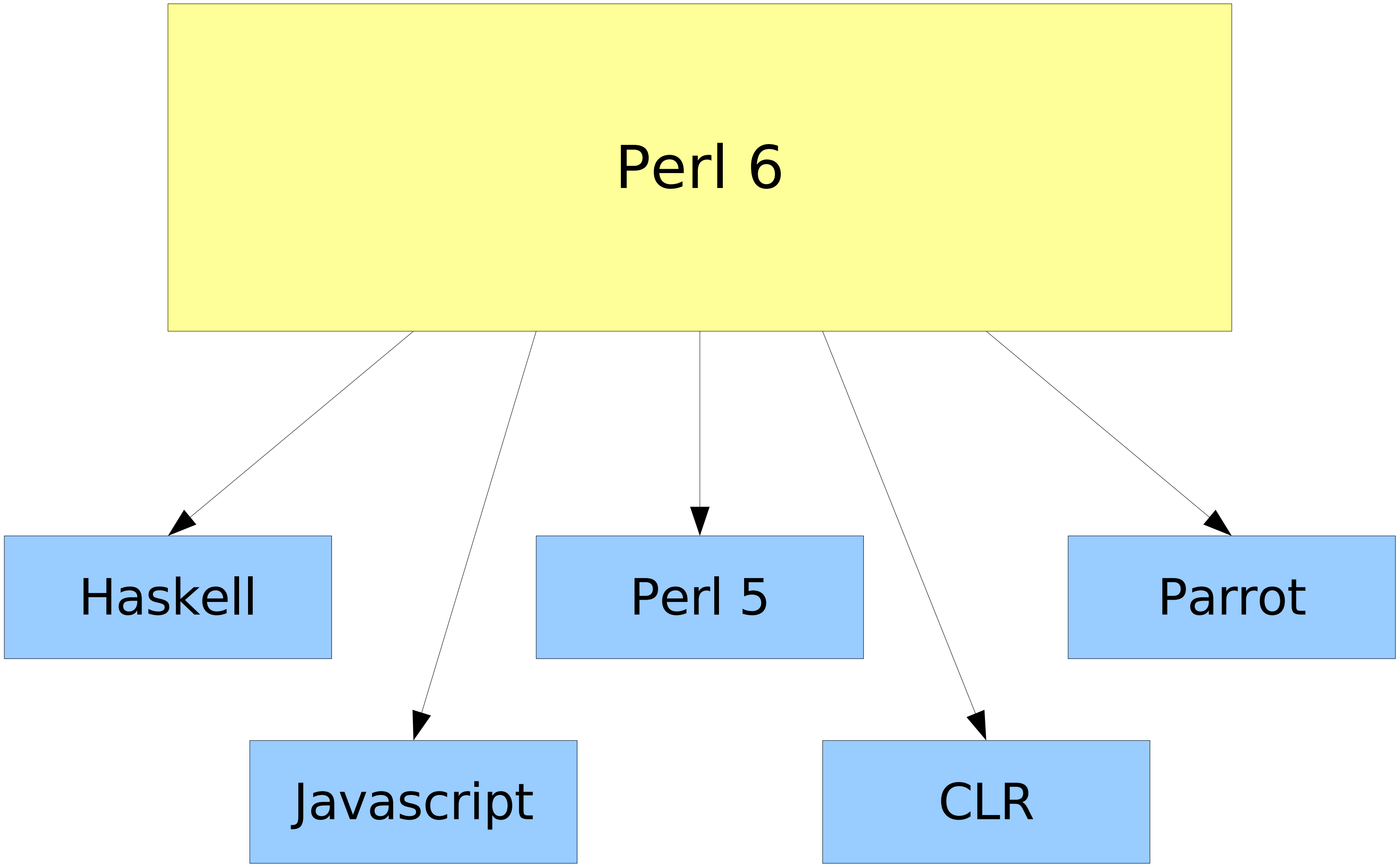
- Oddities

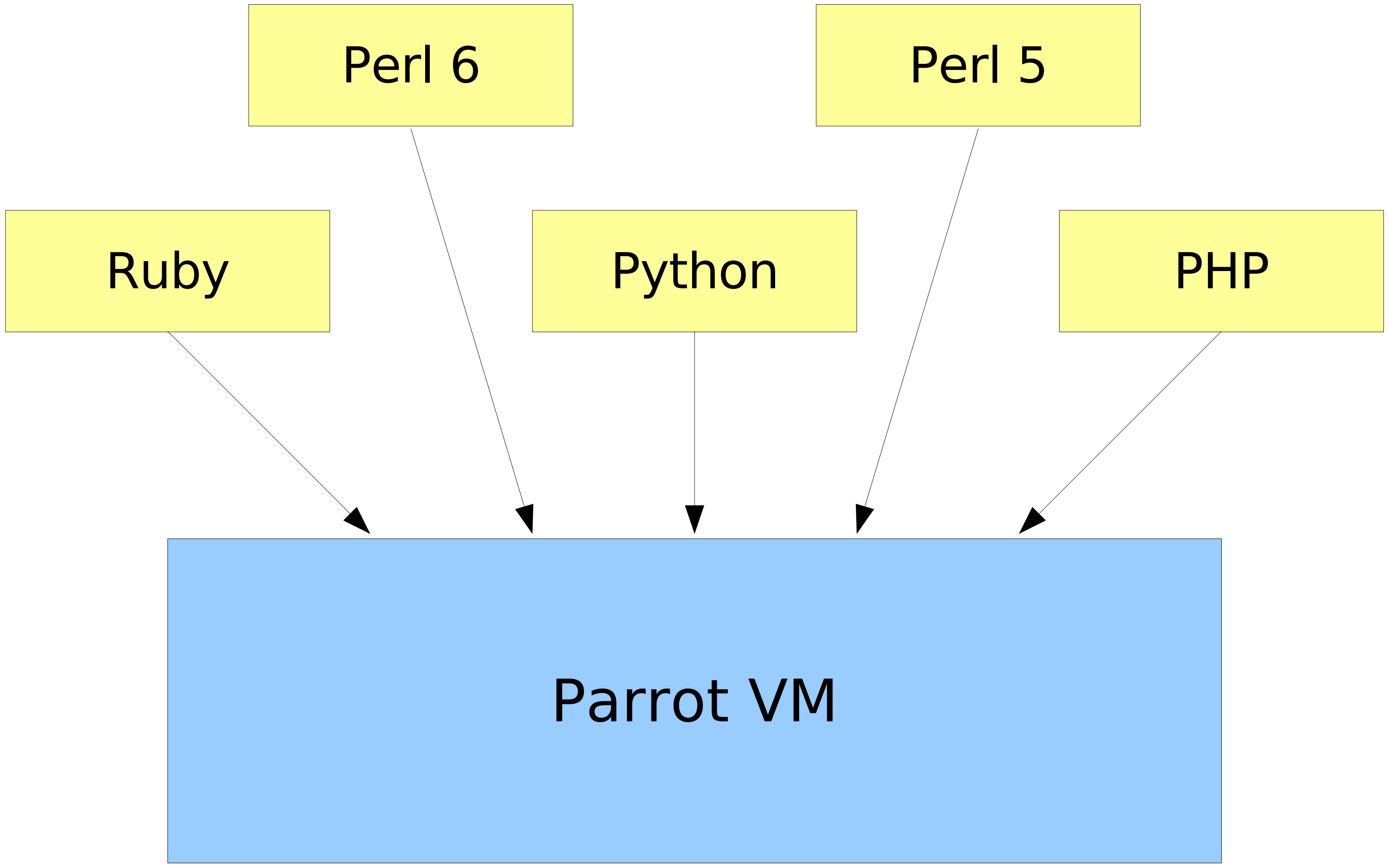
```
    EQ NE GT LT GE LE
```

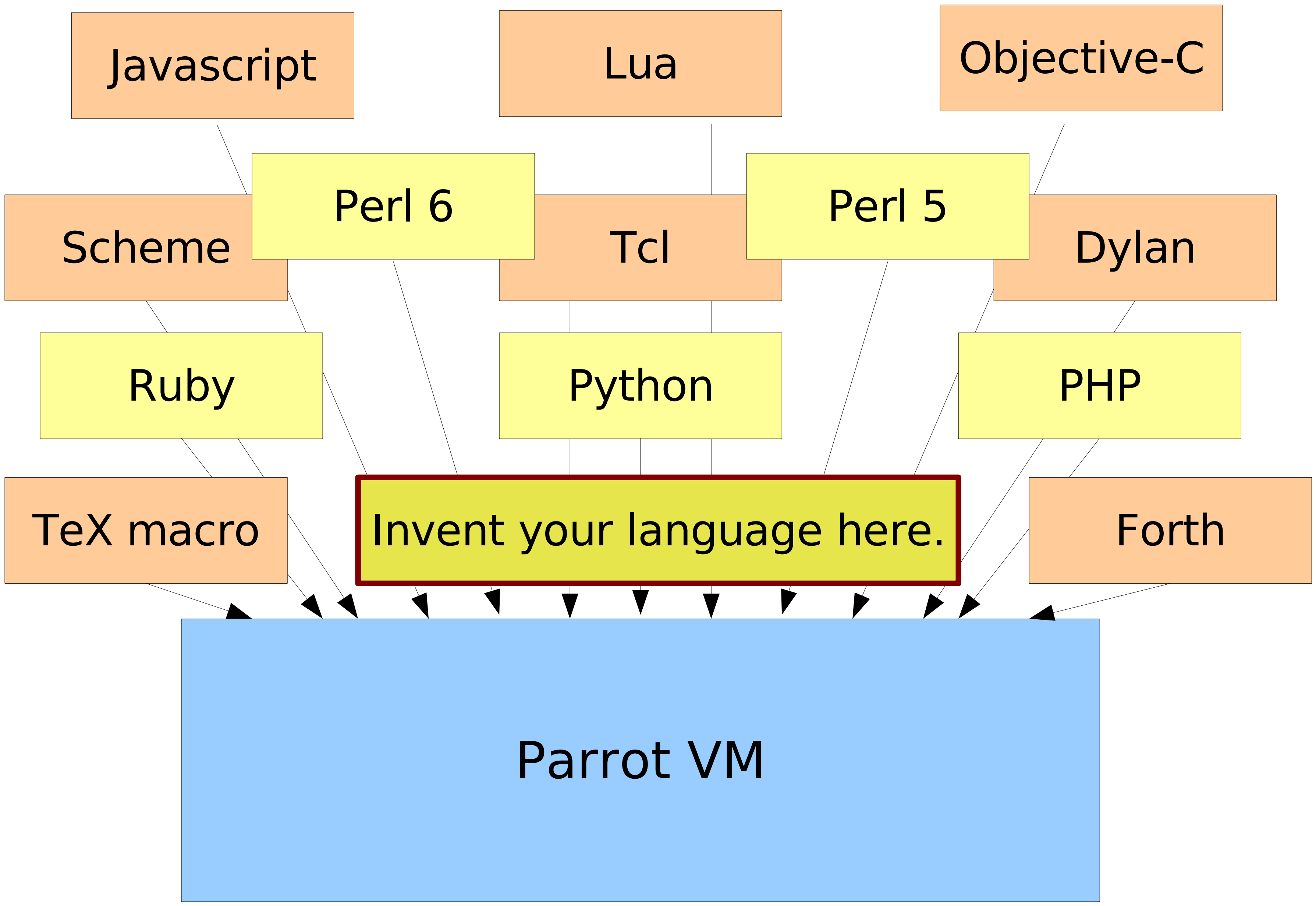
- No lexicals
- No modules or classes

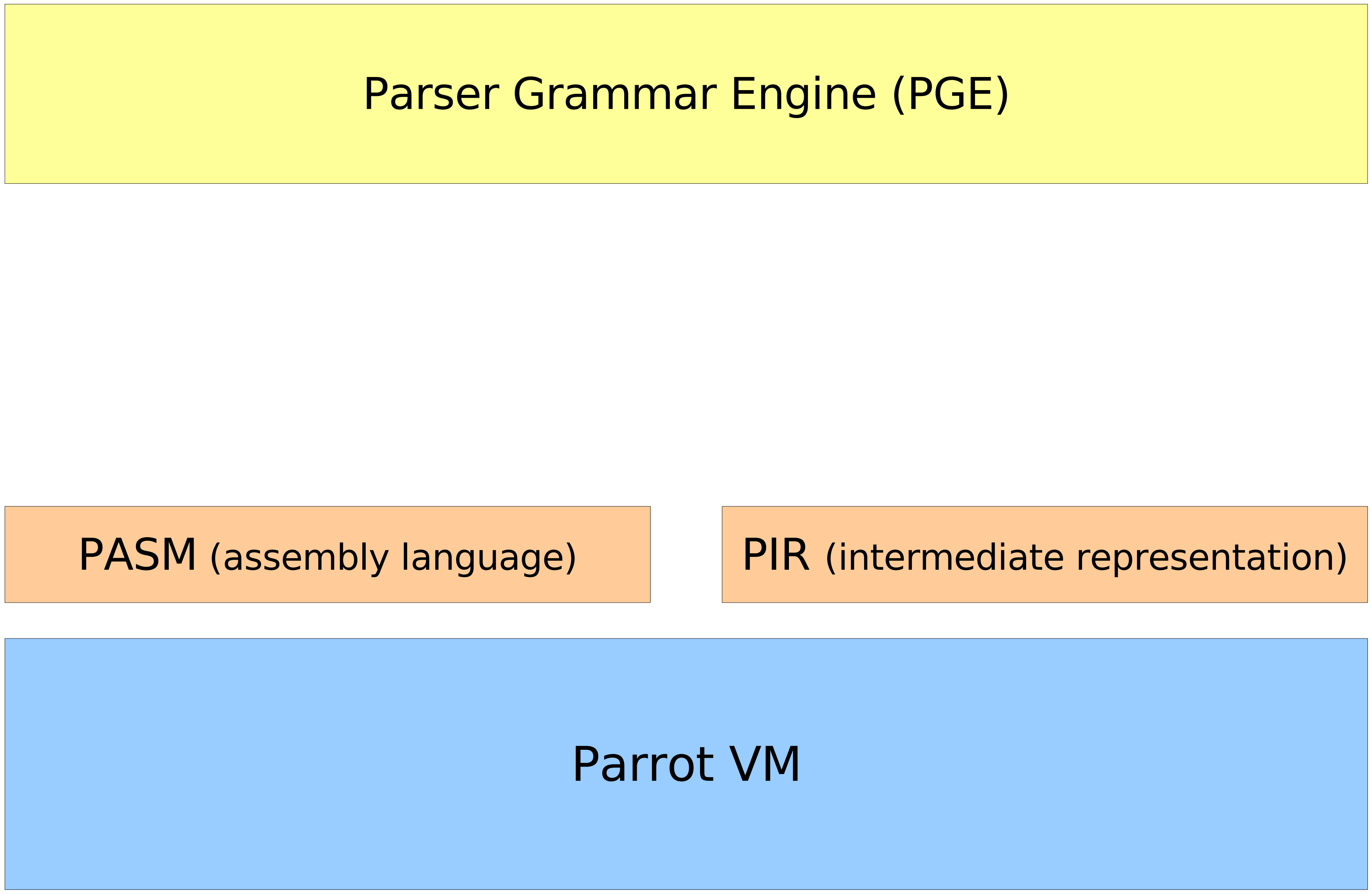
Punie

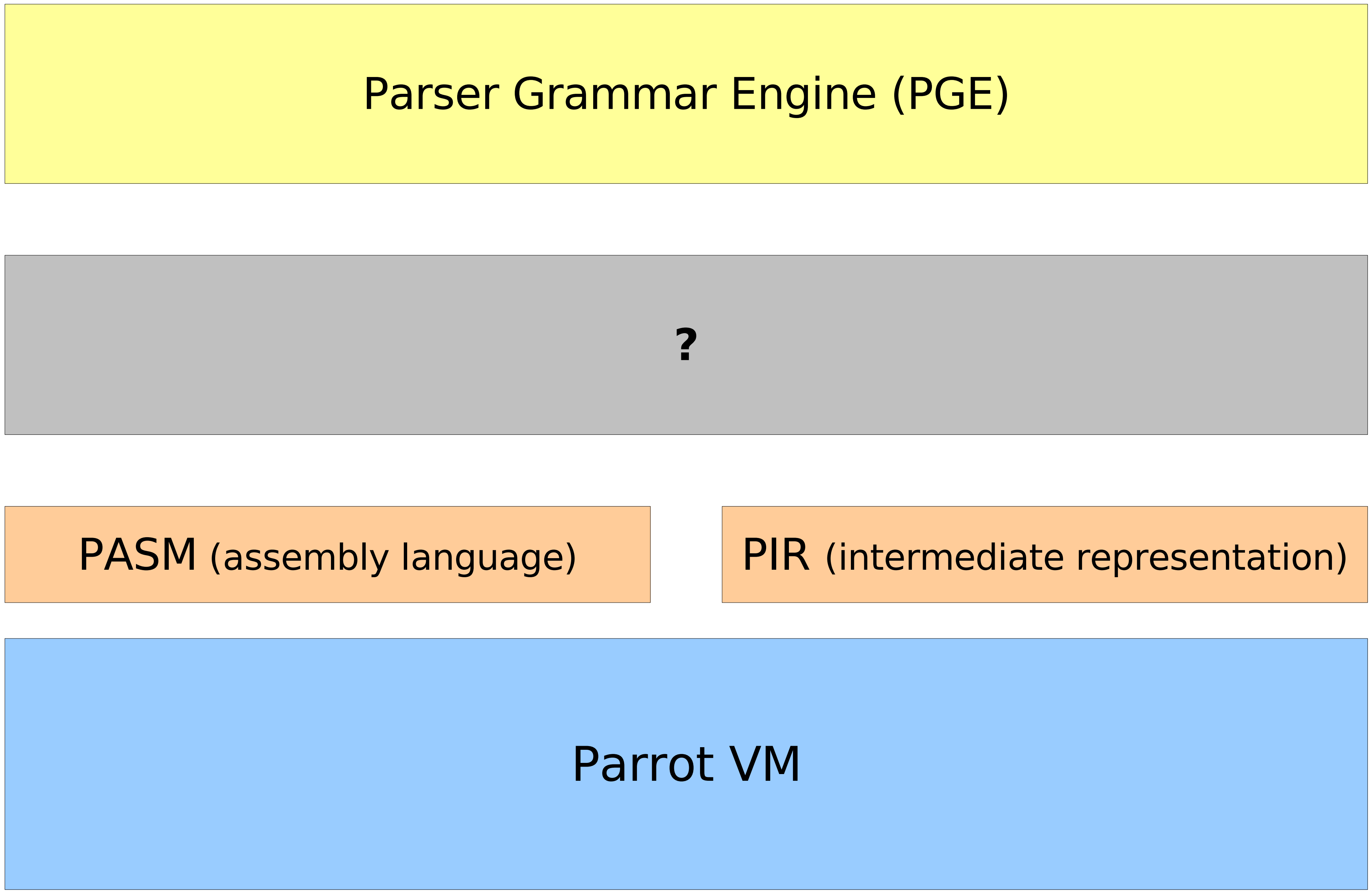
- Test case
- Subset of Perl 5/6
- Non-critical
- More flexibility

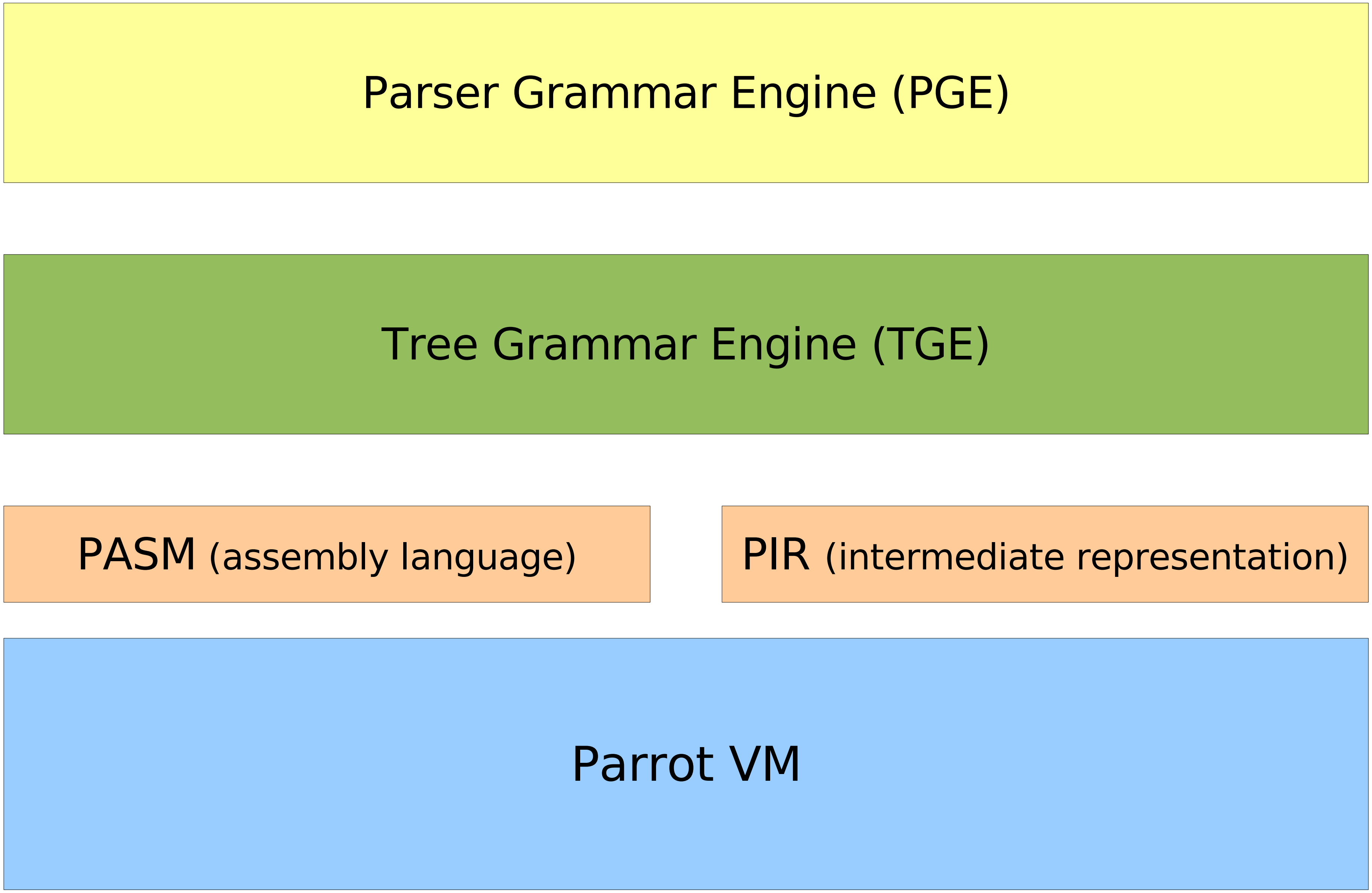

Perl 6
Haskell
Perl 5
Parrot
Javascript
CLR


Perl 6
Perl 5
Ruby
Python
PHP
Parrot VM


Javascript
Lua
Objective-C
Perl 6
Perl 5
Scheme
Tcl
Dylan
Ruby
Python
PHP
TeX macro
Invent your language here.
Forth
Parrot VM


Parser Grammar Engine (PGE)
PASM (assembly language)
PIR (intermediate representation)
Parrot VM


Parser Grammar Engine (PGE)
?
PASM (assembly language)
PIR (intermediate representation)
Parrot VM


Parser Grammar Engine (PGE)
Tree Grammar Engine (TGE)
PASM (assembly language)
PIR (intermediate representation)
Parrot VM

Tree Grammar Engine

- Attribute Grammars

(Early February, 1967)

Peter [Wegner] asked me what I thought about formal semantics, and I said I liked [Ned] Iron's idea of synthesizing an overall meaning from submeanings. I also said that I liked the way other people had combined Irons's approach with a top-down or “recursive-descent” parser...

So Peter asked, “Why can't attributes be defined from the top down as well as from the bottom up?”

A shocking idea! Of course I instinctively replied that it was impossible to go both bottom-up and top-down. But after some discussion I realized that his suggestion wasn't so preposterous after all...

- D. E. Knuth, “The genesis of attribute grammars”

Tree Grammar Engine

- Attribute Grammars
- Minimalist Program

Tree Grammar Engine

- Attribute Grammars
- Minimalist Program
- Borrowing Perlishly

Tree Grammar Engine

- Attribute Grammars
- Minimalist Program
- Borrowing Perlishly
- Attract multiple languages

Tree Grammar Engine

- Attribute Grammars
- Minimalist Program
- Borrowing Perlishly
- Attract multiple languages
- Easy to use

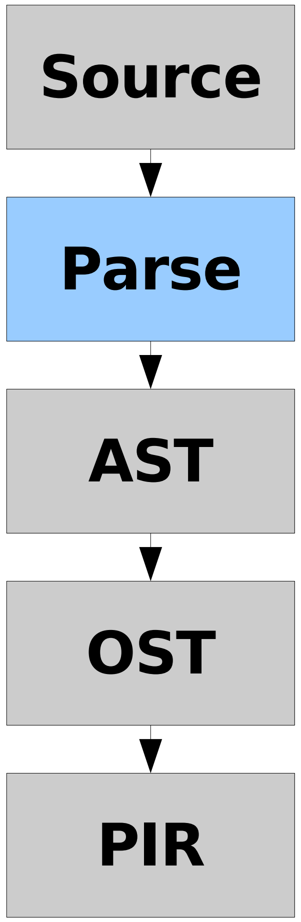
There's an odd misconception in the computing world that writing compilers is hard. This view is fueled by the fact that we don't write compilers very often. People used to think writing CGI code was hard. Well, it is hard, if you do it in C without any tools.

Tree Grammar Engine

- 4 stages

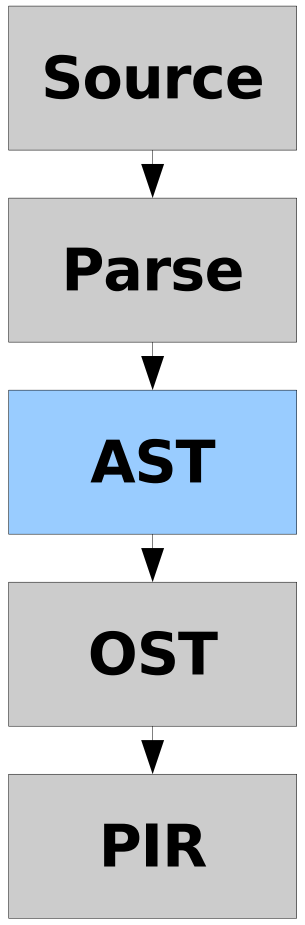
Tree Grammar Engine

- 4 stages
- Parse Tree



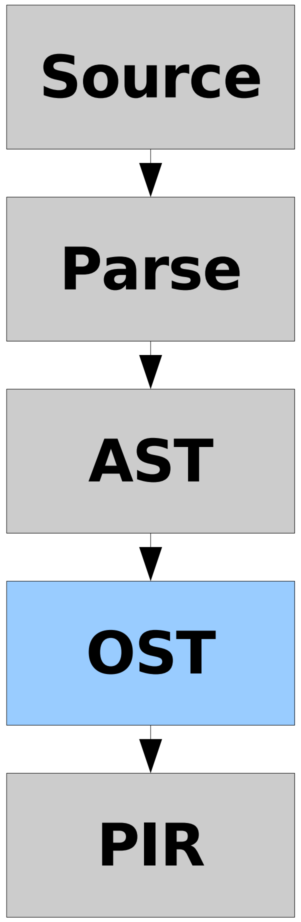
Tree Grammar Engine

- 4 stages
- Parse Tree
- Abstract Syntax Tree



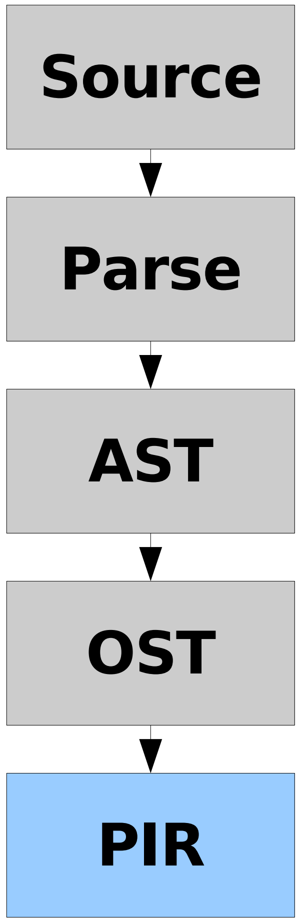
Tree Grammar Engine

- 4 stages
- Parse Tree
- Abstract Syntax Tree
- Opcode Syntax Tree


$$\begin{tikzcd}\text{Source} \arrow[d] \\ \text{Parse} \arrow[d] \\ \text{AST} \arrow[d] \\ \text{OST} \arrow[d] \\ \text{PIR}\end{tikzcd}$$

Tree Grammar Engine

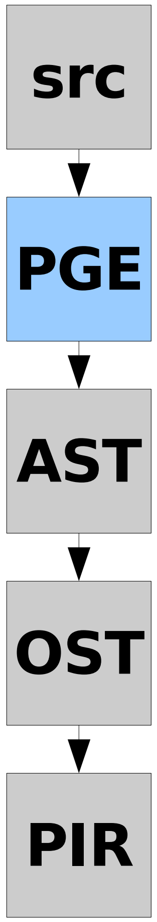
- 4 stages
- Parse Tree
- Abstract Syntax Tree
- Opcode Syntax Tree
- PIR (or bytecode)



Value Transformation

- Simple integer

```
42
```



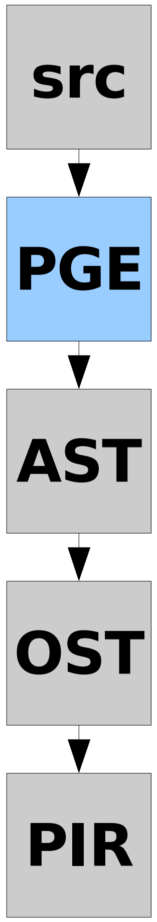
Value Transformation

- Simple integer

```
42
```

- Parser grammar:

```
token integer { \d+ }
```



Value Transformation

- Simple integer

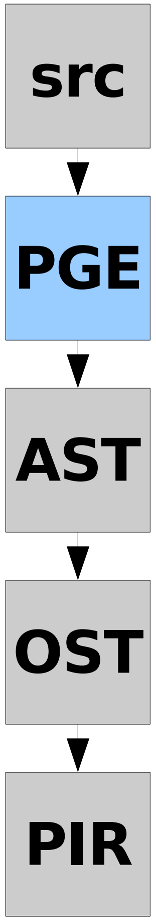
```
42
```

- Parser grammar:

```
token integer { \d+ }
```

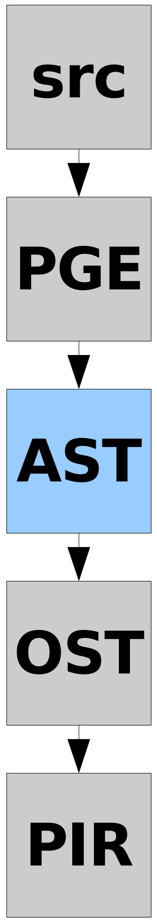
- PGE match tree:

```
<PunieGrammar::integer> =>
    PMC 'PunieGrammar' => "42" @ 0
```



Value Transformation

- AST tree grammar

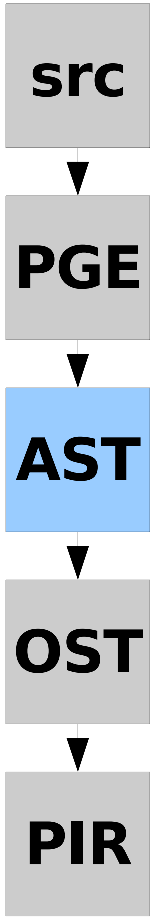


```
transform result (PunieGrammar::integer) :language('PIR') {
    .local pmc result
    result = new 'PAST::Val'

    $S2 = node
    result.'value'($S2)
    result.'valtype'('int')
    .return (result)
}
```

Value Transformation

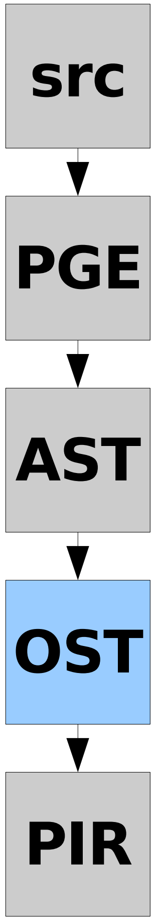
- AST tree grammar
- AST result tree:

```
<PAST::Val> => {
    'source' => '42',
    'pos' => '0',
    'value' => '42',
    'valtype' => 'int',
}
```



Value Transformation

- OST tree grammar

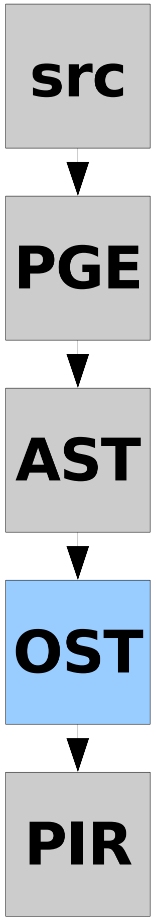


```
transform result (PAST::Val) :language('PIR') {
    .local pmc result
    result = new 'POST::Val'

    $P1 = node.'value'()
    result.'value'($P1)
    $P2 = node.'valtype'()
    result.'valtype'($P2)
    .return (result)
}
```

Value Transformation

- OST tree grammar
- OST result tree:

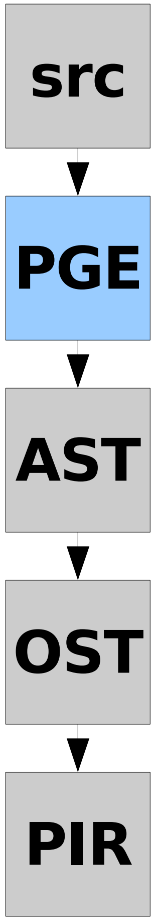
```
<POST::Val> => {
    'source' => '42',
    'pos' => '0',
    'value' => '42',
    'valtype' => 'int',
}
```



Operator Transformation

- Simple statement

```
6 * 9;
```



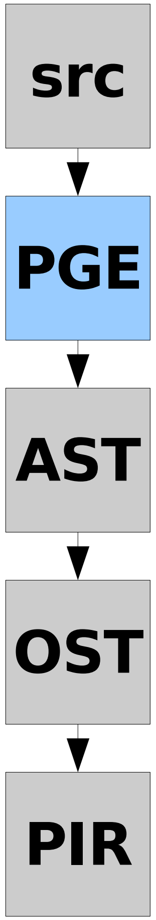
Operator Transformation

- Simple statement

```
6 * 9;
```

- Parser grammar:

```
proto 'infix:*' is tighter('infix:+') { ... }
```



Operator Transformation

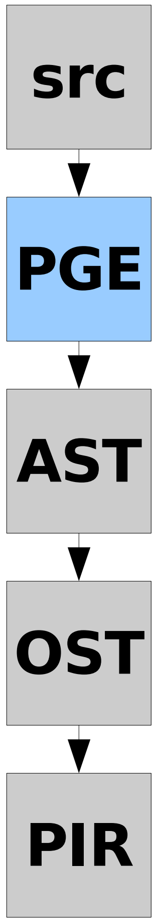
- Simple statement

```
6 * 9;
```

- Parser grammar:

```
proto 'infix:*' is tighter('infix:+') { ... }
```

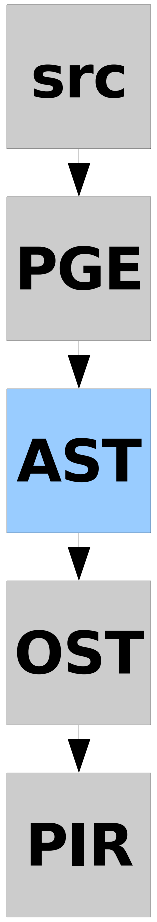
- PGE match tree



```
PMC 'PunieGrammar' => "6 * 9" @ 0 {
    <expr> => PMC 'PGE::Match' => "*" @ 2 {
        <type> => "infix:*"
        [0] => PMC 'PunieGrammar' => "6" @ 0 {
            <PunieGrammar::integer> =>
                    PMC 'PunieGrammar' => "6" @ 0
            <type> => "term:"
        }
        [1] => PMC 'PunieGrammar' => "9" @ 4 {
            <PunieGrammar::integer> =>
                    PMC 'PunieGrammar' => "9" @ 4
            <type> => "term:"
        }
    }
}
```

Operator Transformation

- AST grammar

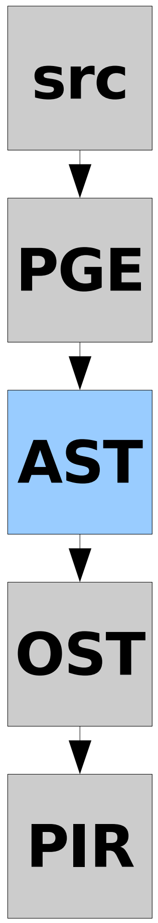


```
transform op (expr) :language('PIR') {
    .local pmc result
    result = new 'PAST::Op'
    result.'clone'(node)
    $S1 = node["type"]
    result.'op'($S1)

    $P1 = node.get_array()
    .local pmc iter
    iter = new Iterator, $P1    # setup iterator for node
    set iter, 0 # reset iterator, begin at start
  iter_loop:
    unless iter, iter_end         # while (entries) ...
      shift $P2, iter             # get entry
      $P3 = tree.get('result', $P2, 'expr')
      if null $P3 goto iter_loop
      result.'add_child'($P3)
      goto iter_loop
  iter_end:

    .return (result)
}
```

```
transform op (expr) :language('PIR') {









      #loop
      $P3 = tree.get('result', $P2, 'expr')

      result.'add_child'($P3)
      # end loop

    .return (result)
}
```

Operator Transformation

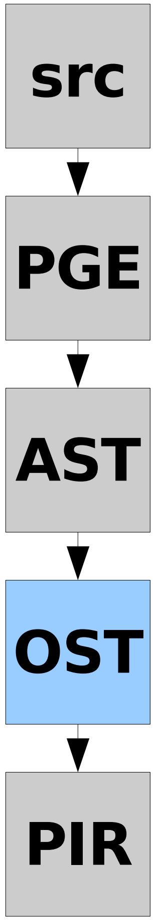
- AST grammar
- AST result tree



```
<PAST::Op> => {
    'source' => '*',
    'pos' => '2',
    'op' => 'infix:*',
    'children' => [
        <PAST::Val> => {
            'source' => '6',
             'pos' => '0',
             'value' => '6',
             'valtype' => 'int',
        }
        <PAST::Val> => {
             'source' => '9',
             'pos' => '4',
             'value' => '9',
             'valtype' => 'int',
        }
    ]
}
```

Operator Transformation

- OST grammar



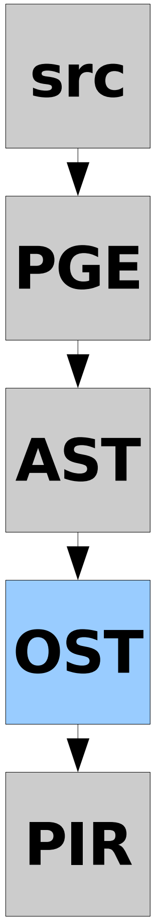
```
transform infix (PAST::Op) :language('PIR') {

      # loop
      $P3 = tree.get('result', $P2)
      #...
      result.'add_child'($P3)
      # end loop

    .return (result)
}
```

```
transform infix (PAST::Op) :language('PIR') {
    newops = new 'POST::Ops'

      ...
      # loop
      $P3 = tree.get('result', $P2)

      $S1 = typeof $P3
      if $S1 == 'POST::Ops' goto complex_result
      if $S1 == 'POST::Val' goto create_tmp
        childop.'add_child'($P3)
      ...

      # end loop


    newops.'add_child'(childop)

    .return (newops)
}
```

```
if $S1 == 'POST::Val' goto create_tmp


create_tmp:
  # Create a temp variable
  $P4 = new 'POST::Var'

  $P5 = $P4.new_dummy()
  newops.'add_child'($P5)
  childop.'add_child'($P4)

  # Assign the value node to the variable
  $P7 = new 'POST::Op'

  $P7.'op'('set')
  $P7.'add_child'($P4) # the first argument is the variable
  $P7.'add_child'($P3) # the second argument is the value
  newops.'add_child'($P7)

# end loop
# e.g. 1 + 2 + 3,
#       or $y = 1 + 1 && 11 + 11

if $S1 == 'POST::Ops' goto complex_result


complex_result:
  $P1 = $P3.tmpvar()
  childop.'add_child'($P1)
  newops.'add_child'($P3)

# end loop
```

Operator Transformation

- OST grammar
- OST result tree

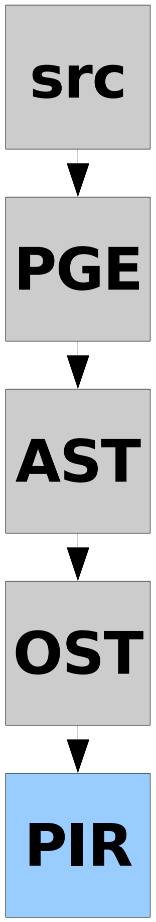


```
<POST::Ops> => {
    'source' => '*',
    'pos' => '2',
    'children' => [
        <POST::Op> => {
            'source' => '*',
            'pos' => '2',
            'op' => 'new',
            'children' => [
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P1',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Val> => {
                    'source' => undef,
                    'pos' => '0',
                    'value' => '71',
                    'valtype' => 'int',
                }
            ]
        }
        <POST::Op> => {
            'source' => '*',
            'pos' => '2',
            'op' => 'new',
            'children' => [
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P2',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Val> => {
                    'source' => undef,
                    'pos' => '0',
                    'value' => '71',
                    'valtype' => 'int',
                }
            ]
        }
        <POST::Op> => {
            'source' => '*',
            'pos' => '2',
            'op' => 'set',
            'children' => [
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P2',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Val> => {
                    'source' => '6',
                    'pos' => '0',
                    'value' => '6',
                    'valtype' => 'int',
                }
            ]
        }
        <POST::Op> => {
            'source' => '*',
            'pos' => '2',
            'op' => 'new',
            'children' => [
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P3',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Val> => {
                    'source' => undef,
                    'pos' => '0',
                    'value' => '71',
                    'valtype' => 'int',
                }
            ]
        }
        <POST::Op> => {
            'source' => '*',
            'pos' => '2',
            'op' => 'set',
            'children' => [
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P3',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Val> => {
                    'source' => '9',
                    'pos' => '4',
                    'value' => '9',
                    'valtype' => 'int',
                }
            ]
        }
        <POST::Op> => {
            'source' => undef,
            'pos' => '0',
            'op' => 'mul',
            'children' => [
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P1',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P2',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
                <POST::Var> => {
                    'source' => '*',
                    'pos' => '2',
                    'varname' => '$P3',
                    'hllname' => undef,
                    'vartype' => undef,
                    'scope' => undef,
                }
            ]
        }
    ]
}
```

```
<POST::Op> => {
    'source' => undef,
    'pos' => '0',
    'op' => 'mul',
    'children' => [
        <POST::Var> => {
            'source' => '*',
            'pos' => '2',
            'varname' => '$P1',
            'hllname' => undef,
            'vartype' => undef,
            'scope' => undef,
        }
        <POST::Var> => {
            'source' => '*',
            'pos' => '2',
            'varname' => '$P2',
            'hllname' => undef,
            'vartype' => undef,
            'scope' => undef,
        }
        <POST::Var> => {
            'source' => '*',
            'pos' => '2',
            'varname' => '$P3',
            'hllname' => undef,
            'vartype' => undef,
            'scope' => undef,
        }
    ]
 }
```

Operator Transformation

- PIR tree grammar

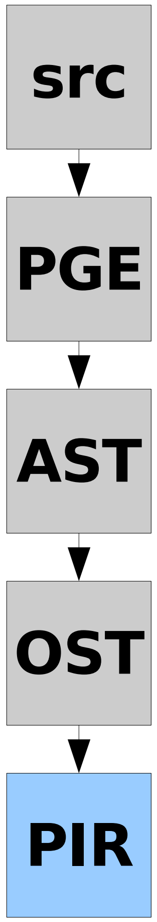


```
transform result (POST::Op) :language('PIR') {
    .local string output
    ...
    opname = node.op()
    output = "    " . opname
    output .= " "

     # loop
      $S3 = tree.get('result', $P2)
      ...
      output .= $S3

      # end loop

    output .= "\n"
    .return (output)
}
```

Operator Transformation

- PIR tree grammar
- PIR output

```
.sub _main :main
    new $P1, .Undef
    new $P2, .Undef
    set $P2, 6
    new $P3, .Undef
    set $P3, 9
    mul $P1, $P2, $P3
.end
```



Tree Grammar Engine

- Simple steps
- Elegant
- Hide Complexity
- Impossible

Revelations

- TMTOWTDI

Revelations

- TMTOWTDI
- Tools shape craft

Revelations

- TMTOWTDI
- Tools shape craft
- Change the universe

Questions?

- Further Reading
  - *http://parrotcode.org/docs/compiler_tools.html*
  - Knuth, D. E. (1990) “The genesis of attribute grammars.” *Proceedings of the international conference on Attribute grammars and their applications*, 1–12.
  - Chomsky, Noam (1995). The Minimalist Program. MIT Press.