Parrot VM

Allison Randal
Parrot Foundation &
O'Reilly Media, Inc.
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.
Parrot VM

- PHP
- Ruby
- Perl 5
- Perl 6
- Python
Invent your language here.
Dynamic Languages

Runtime vs. compile-time
Extend code (eval, load)
Define classes
Alter type system
Higher-order functions
Closures, Continuations, Coroutines
Why?

Revolution
Powerful tools
Portability
Interoperability
Innovation
Register-based

Stack operations
Register-based

Stack operations

14
Register-based

Stack operations
Register-based

Stack operations

```
add 9
  14
```
Register-based

Stack operations
Register-based

Stack operations
Register operations
Register-based

Stack operations
Register operations

9 14
add
Register-based

Stack operations
Register operations
Register-based

Stack operations
Register operations
Fewer instructions
Hardware registers
Register spilling
Flexible register sets
Continuation Passing Style

Stack-based control flow
Continuation Passing Style

Stack-based control flow
Continuation Passing Style

Stack-based control flow
Continuation Passing Style

Stack-based control flow
Continuation Passing Style

Stack-based control flow

return addr
Continuation Passing Style

Stack-based control flow
Continuation Passing Style

Stack-based control flow

Continuation-based control flow
Continuation Passing Style

Stack-based control flow

Continuation-based control flow
Continuation Passing Style

Stack-based control flow

Continuation-based control flow
Continuation Passing Style

Stack-based control flow
Continuation-based control flow

Context:
- main
- foo
- bar
Continuation Passing Style

Stack-based control flow

Continuation-based control flow
Continuation Passing Style

Stack-based control flow
Continuation-based control flow
Deeply nested contexts
Tail recursion
PASM

Assembly language
Simple syntax
add I0, I1, I2
Human-readable bytecode
Syntactic sugar

$\text{IO} = \text{I1} + \text{I2}$

Named variables

.local int myvar

$\text{IO} = \text{myvar} + 5$

Sub and method calls

result = object.'method'($\text{IO}$)
NQP

Not Quite P(HP|ython|erl|uby)

Lightweight language

\$a \ := \ 1; \\
print(a, \ "\n");

Compiler tools

\$past \ := \ PAST::Op.new( \ :name('printnl') \ );
Parser Grammar Engine

Regular expressions
Recursive descent
Operator precedence parser
HLLCompiler

Base library
Quick start
Common features
Pipp

Download

http://www.parrot.org

Build

$ perl Configure.PL
$ make test

Language

$ cd languages/pipp
$ make test
hello.php

<?php
echo "Hello, World!\n";
?>

Run

$ parrot pipp.pir hello.php
$ pipp hello.php
```plaintext
$P1 = new ['PCT';'HLLCompiler']
$P1.'language'('Pipp')
$P1.'parsegrammar'(['Pipp';'Grammar'])
$P1.'parseactions'(['Pipp';'Grammar';'Actions'])
```
Parser

rule argument_list {
    [ <expression> [',', <expression>]* ]?
    { * }
}
Transform to AST

method echo_statement($/) {
    my $past := $( $<argument_list> );
    $past.name( 'echo' );
    make $past;
}
Value Transformation

42
value transformation

42

Parser grammar
rule “integer”

d+

token integer { \d+ }
Value Transformation

Parser grammar rule “integer”

42
\d+

Parse tree

<integer>
value: 42
Parser grammar rule “integer”

\d+

Parse tree

<integer>
value: 42

Transform rule

integer

method integer($/) {...}
Value Transformation

Parser grammar rule “integer”

\d+

Parse tree

<integer>
value: 42

Transform rule

integer

AST node

<\PAST::Val>
value: 42
returns: Integer
Value Transformation

42 → \d+ → <integer> value: 42 → integer

AST node

\textless\text{PAST::Val}\textgreater
value: 42
returns: Integer

Transform rule

PAST::Val

\texttt{ost = self.as\_post(\text{ast})}
Value Transformation

Parser grammar rule “integer”

\d+

Parse tree

<integer>
value: 42

Transform rule

integer

AST node

<PAST::Val>
value: 42
returns: Integer

Transform rule

PAST::Val

OST node

<POST::Op>
result: 1
Value Transformation

Parser grammar rule “integer”

\d+ → <integer> value: 42 integer

Parse tree

Transform rule

AST node

<PAST::Val>
value: 42
returns: Integer

Transform rule

PAST::Val

OST node

<POST::Op>
result: 1

Transform rule

POST::Val

self.pir(ost)
Value Transformation

Parser grammar rule “integer”

\d+

Parse tree

<integer>
value: 42

Transform rule

integer

AST node

<PAST::Val>
value: 42
returns: Integer

Transform rule

PAST::Val

OST node

<POST::Op>
result: 1

Transform rule

POST::Val

42
Operator Transformation

6 * 9
Operator Transformation

Parser grammar
OPP rule

6 * 9 → infix:*  

proto infix:* is looser(prefix:+) {...}
Operator Transformation

Parser grammar
OPP rule

Parse tree

6 * 9

infix:*

type: 'infix:*'

<integer>
value: 6

<integer>
value: 9
Operator Transformation

Parse tree

<expr>
  type: 'infix:*'

<integer>
  value: 6

<integer>
  value: 9
Operator Transformation

Parse tree

<expr>
  type: 'infix:*'

Transform rule

expr

<integer>
  value: 6

<integer>
  value: 9
Operator Transformation

Parse tree

<expr>
type: 'infix:*'

Transform rule

expr

AST tree

<PAST::Op>
name: infix:*

<integer>
value: 6

<integer>
value: 9

<PAST::Val>
value: 6
returns: Integer

<PAST::Val>
value: 9
returns: Integer
Operator Transformation

AST tree

```
<PAST::Op>
name: infix:*  

<PAST::Val>
value: 6  
returns: Integer

<PAST::Val>
value: 9  
returns: Integer
```
Operator Transformation

AST tree

\[
\begin{align*}
\text{<PAST::Op>} & \quad \text{name: infix:*} \\
\begin{align*}
\text{<PAST::Val>} & \quad \text{value: 6} \\
\text{returns: Integer} \\
\end{align*} & \quad \begin{align*}
\text{<PAST::Val>} & \quad \text{value: 9} \\
\text{returns: Integer} \\
\end{align*}
\end{align*}
\]

Transform rule

PAST::Op
Operator Transformation

AST tree

*PAST::Op*
- name: infix: *

*PAST::Val*
- value: 6
  - returns: Integer
- value: 9
  - returns: Integer

Transform rule

*PAST::Op*

OST tree

*POST::Ops*
- result: 6
- result: 9

*POST::Op*
- name: n_mul

**variable setup**

*POST::Ops*
- result: $P1$
- result: 6
- result: 9
Operator Transformation

.sub _main :main
    new $P1, 'Integer'
    new $P2, 'Integer'
    set $P2, 6
    new $P3, 'Integer'
    set $P3, 9
    mul $P1, $P2, $P3
.end
Examples

In the Parrot distribution:

```
examples/tutorial/*.pir
```
Further Reading

“Continuations and advanced flow control” by Jonathan Bartlett

“The case for virtual register machines” by Brian Davis, et al.
<http://portal.acm.org/citation.cfm?id=858575>

Pipp project site
http://www.pipp.org