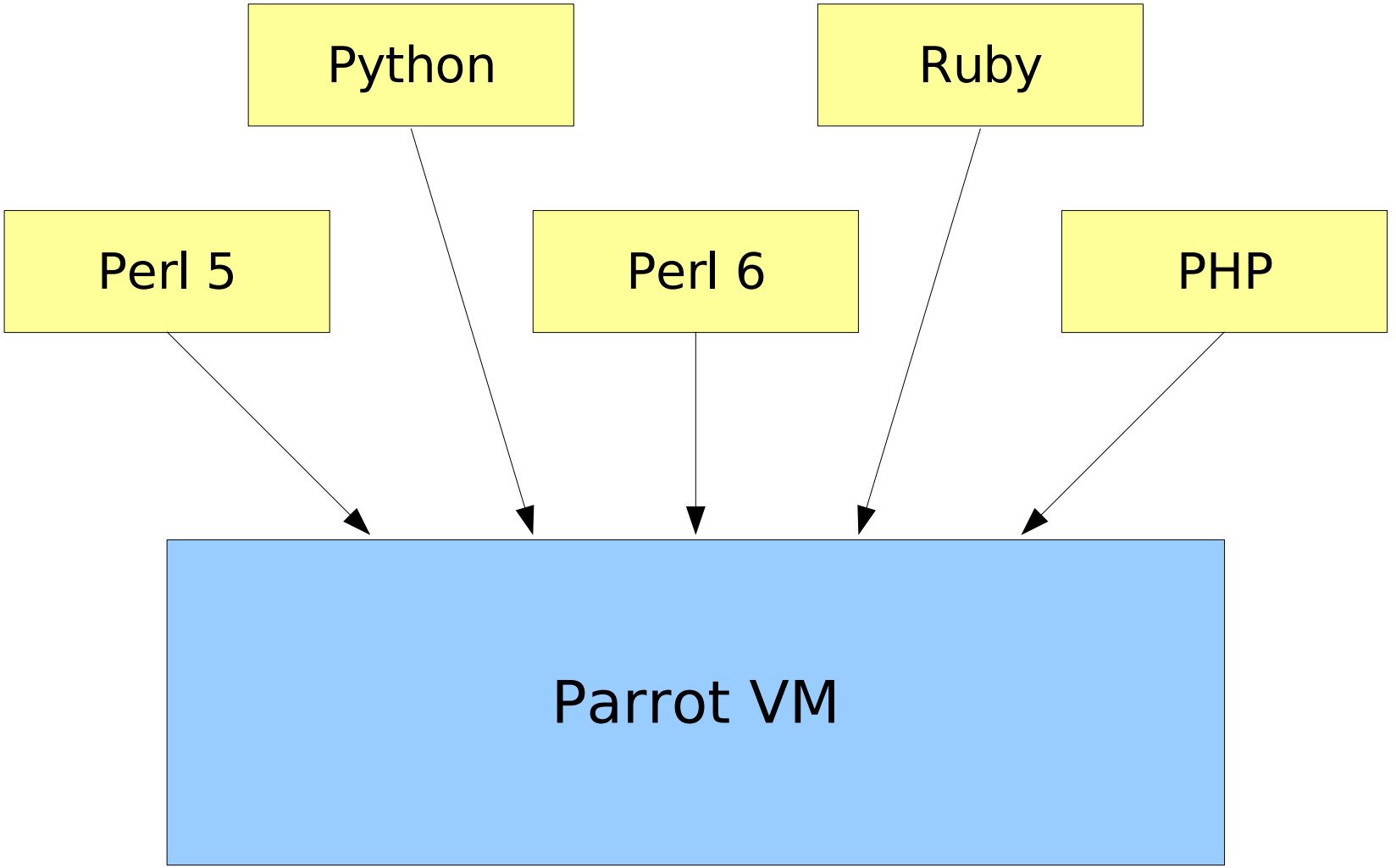
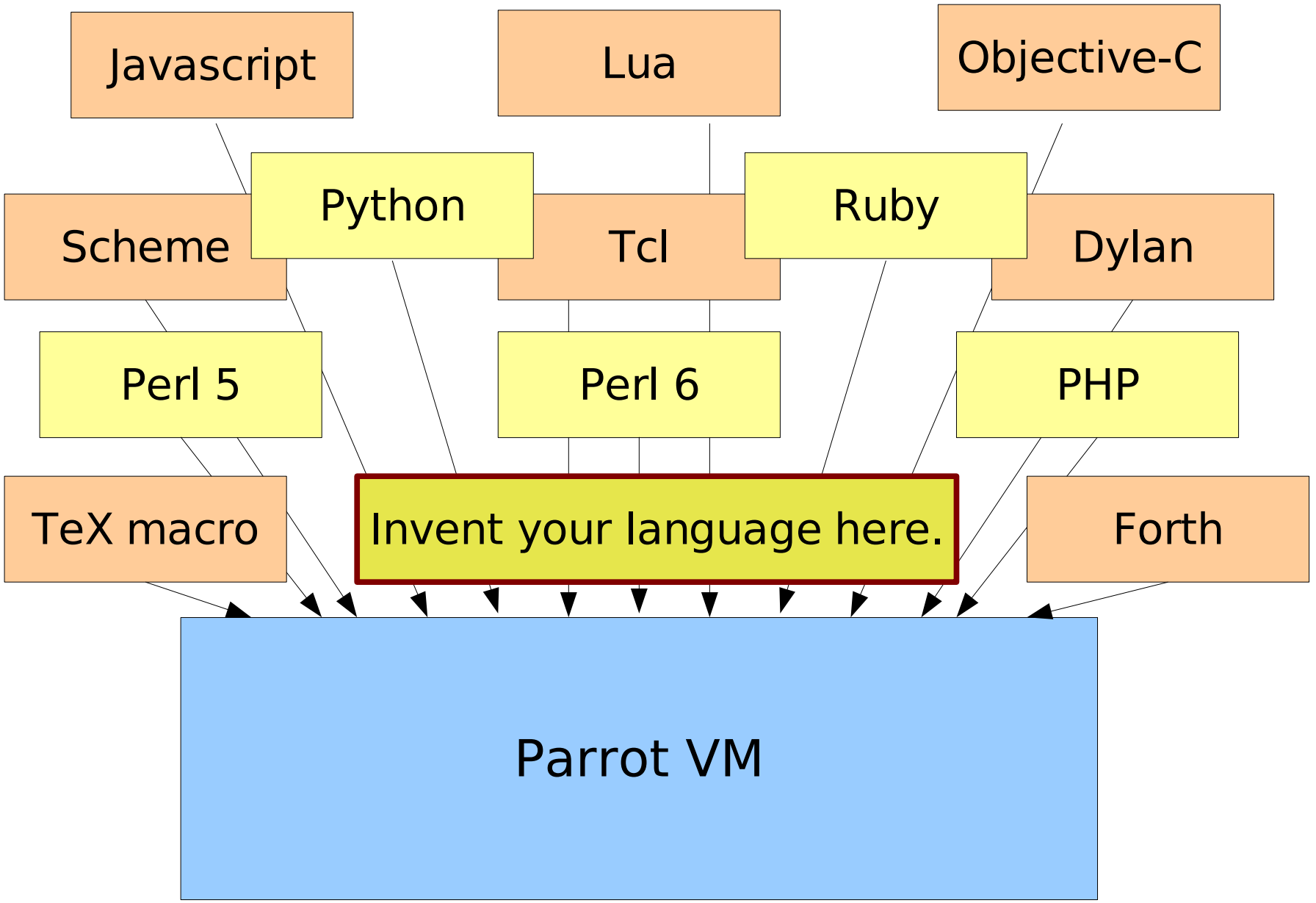
Parrot

Allison Randal
The Perl 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.


Python
Ruby
Perl 5
Perl 6
PHP
Parrot VM


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

Dynamic Languages

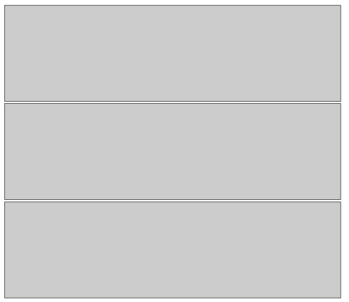
- Runtime vs. compile-time
- Extend code (eval, load)
- Define classes
- Alter type system
- Higher-order functions
- Closures, Continuations, Coroutines

Why?

- Revolution
- Powerful tools
- Portability
- Interoperability
- Drive innovation

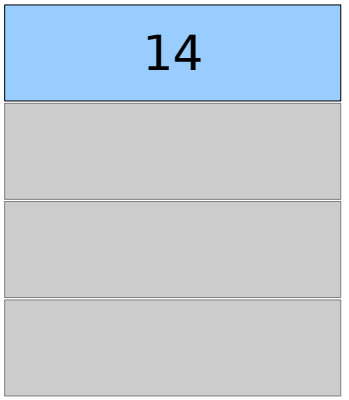
Register-based

- Stack operations



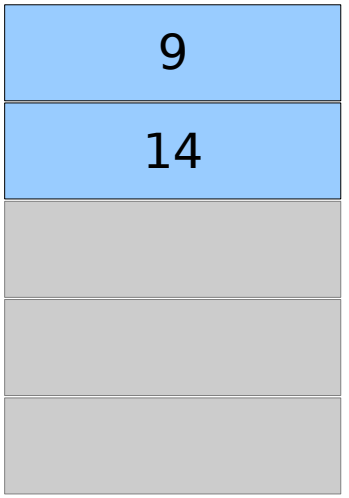
Register-based

- Stack operations



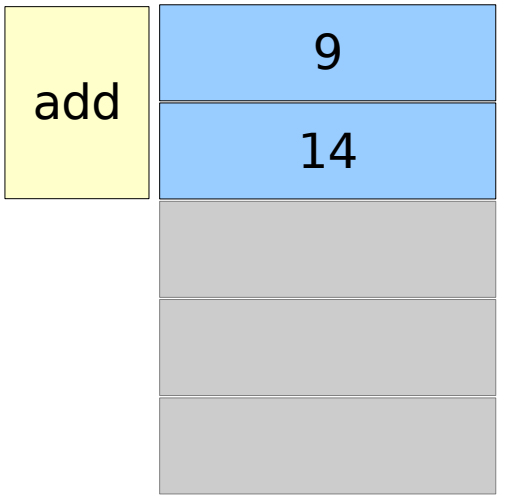
Register-based

- Stack operations



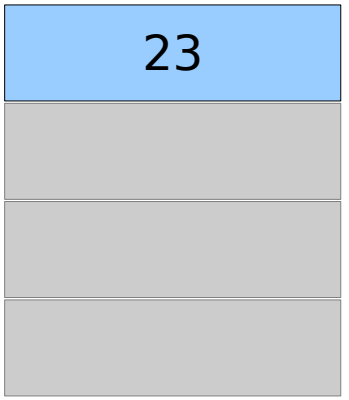
Register-based

- Stack operations



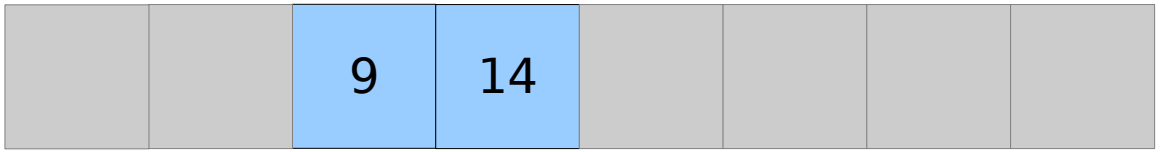
Register-based

- Stack operations



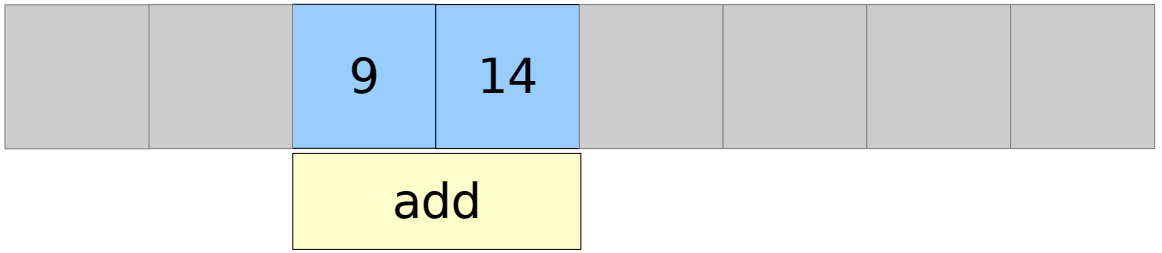
Register-based

- Stack operations
- Register operations



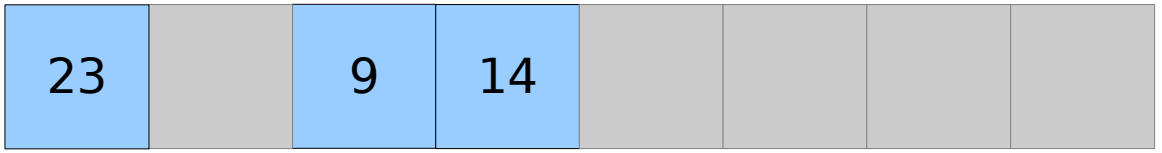
Register-based

- Stack operations
- Register operations



Register-based

- Stack operations
- Register operations



Register-based

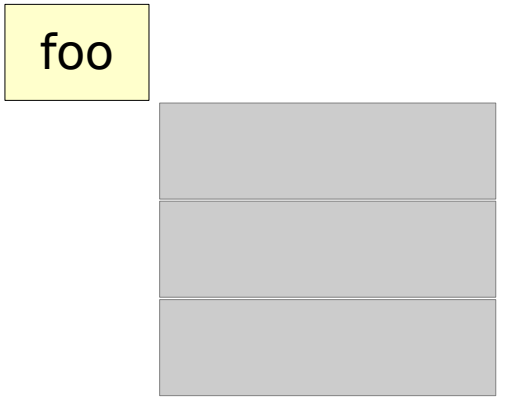
- Stack operations
- Register operations
- Fewer instructions
- Hardware registers
- Register spilling
- Flexible register sets

Continuation Passing Style

- Stack-based control flow

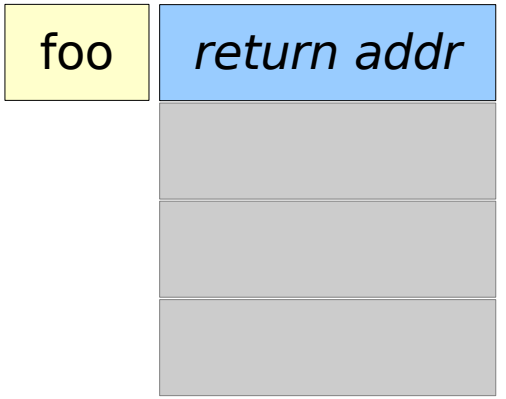
Continuation Passing Style

- Stack-based control flow



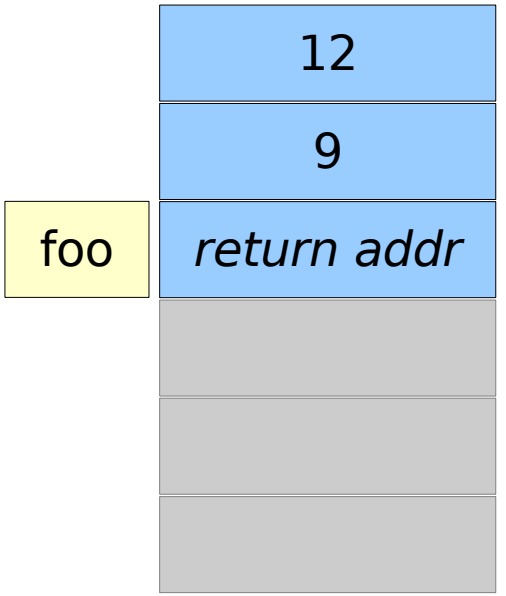
Continuation Passing Style

- Stack-based control flow



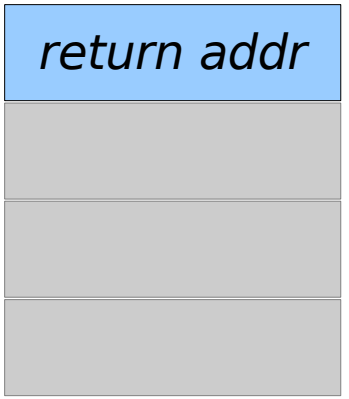
Continuation Passing Style

- Stack-based control flow



Continuation Passing Style

- Stack-based control flow

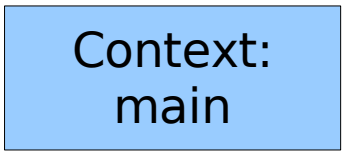


Continuation Passing Style

- Stack-based control flow

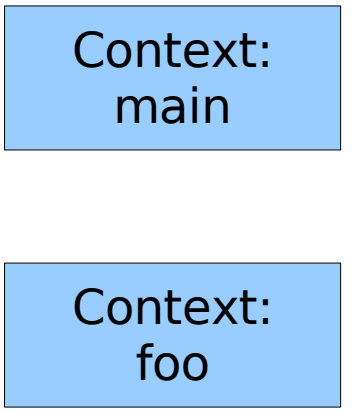
Continuation Passing Style

- Stack-based control flow
- Continuation-based control flow



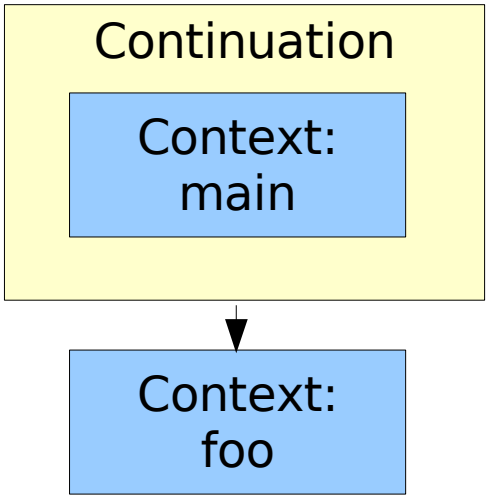
Continuation Passing Style

- Stack-based control flow
- Continuation-based control flow



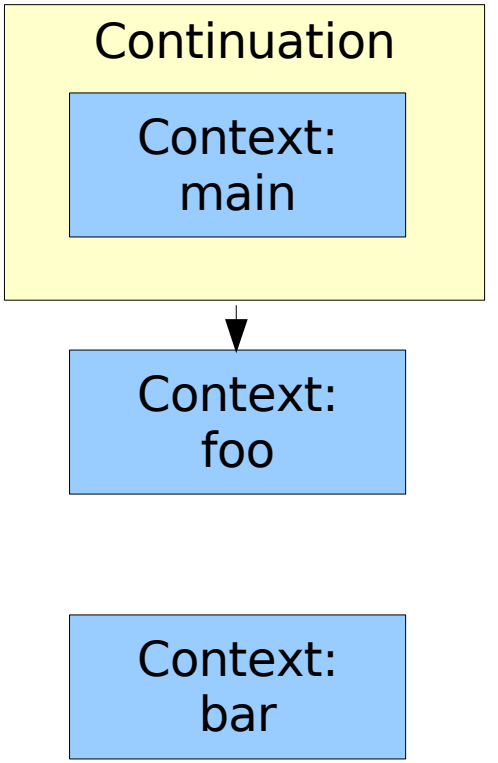
Continuation Passing Style

- Stack-based control flow
- Continuation-based control flow



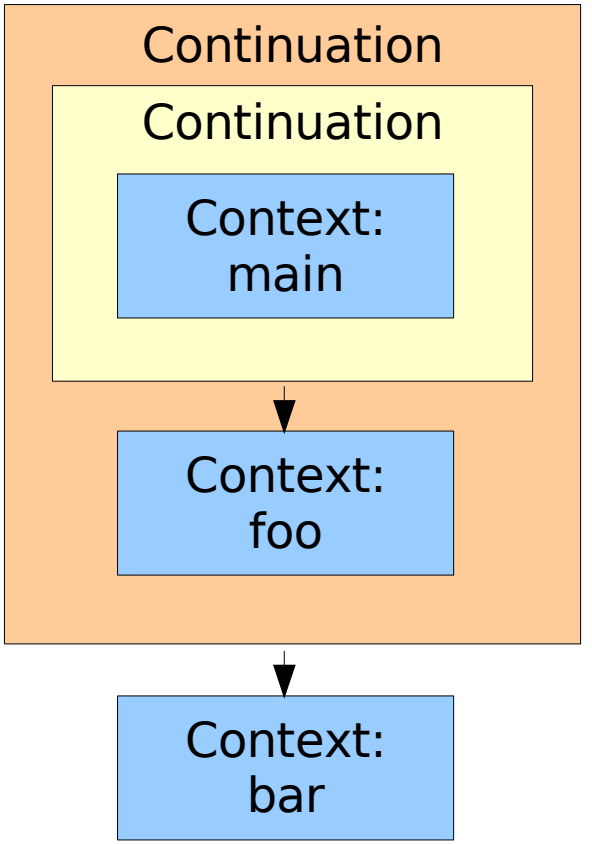
Continuation Passing Style

- Stack-based control flow
- Continuation-based control flow



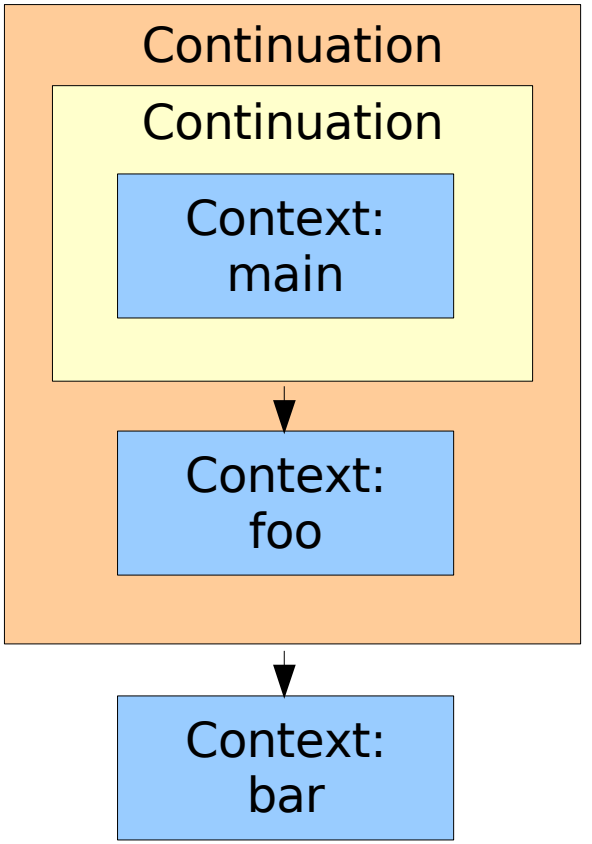
Continuation Passing Style

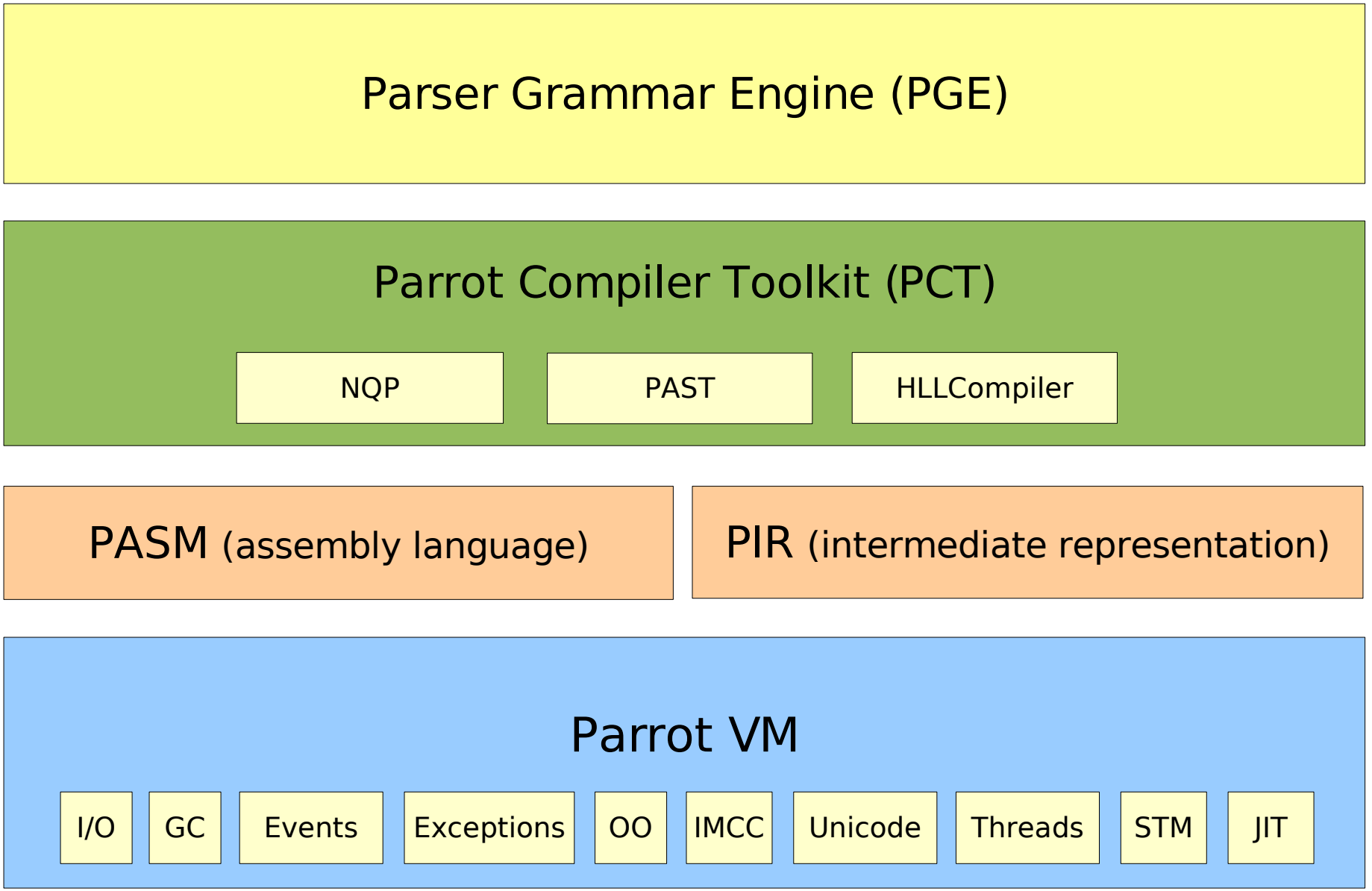
- Stack-based control flow
- Continuation-based control flow



Continuation Passing Style

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




Parser Grammar Engine (PGE)
Parrot Compiler Toolkit (PCT)
NQP
PAST
HLLCompiler
PASM (assembly language)
PIR (intermediate representation)
Parrot VM
I/O
GC
Events
Exceptions
OO
IMCC
Unicode
Threads
STM
JIT

PASM

- Assembly language
- Simple syntax

```
add I0, I1, I2
```

- Human-readable bytecode

PIR

- Syntactic sugar

```
$I0 = $I1 + $I2
```

- Named variables

```
.local int myvar
$I0 = myvar + 5
```

- Sub and method calls

```
result = object.'method'($I0)
```

NQP

- Not Quite P(erl|ython|HP|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

Pynie

- Download

```
http://www.parrotcode.org
```

- Build

```
$ perl Configure.PL
$ make test
```

- Language

```
$ cd languages/pynie
$ make test
```

Pynie

- hello.py

```
print "Hello, World!"
```

- Run

```
$ parrot pynie.pir hello.py
```

pynie.pir

- 67 lines
- Half documentation

```
c = hllcompiler.new()
c.language('Pynie')
c.parsegrammar('Pynie::Grammar')
c.parseactions('Pynie::Grammar::Actions')
```

Grammar.pg

- Parser

```
token stmt_list {
  <simple_stmt> [ ';' <simple_stmt> ]* ';'?
  {*}
}
```

- Familiar?

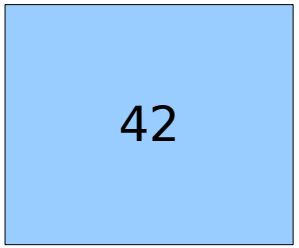
```
stmt_list ::=
  simple_stmt (";" simple_stmt)* [";"]
```

Actions.pct

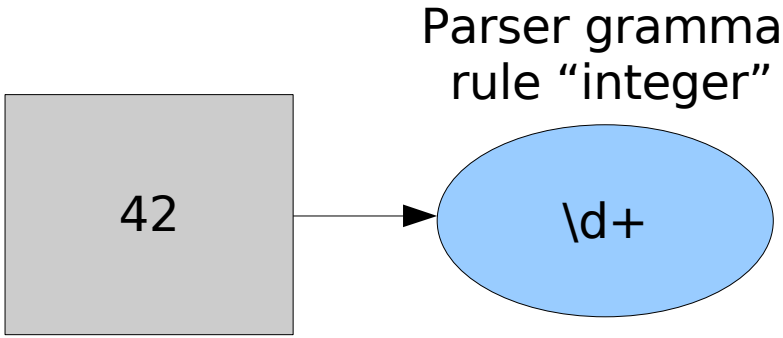
- Transform to AST

```
method identifier($/) {
    make PAST::Var.new( :name( ~$/ ),
                        :scope('package'),
                        :node($/)
                      );
}
```

Value Transformation

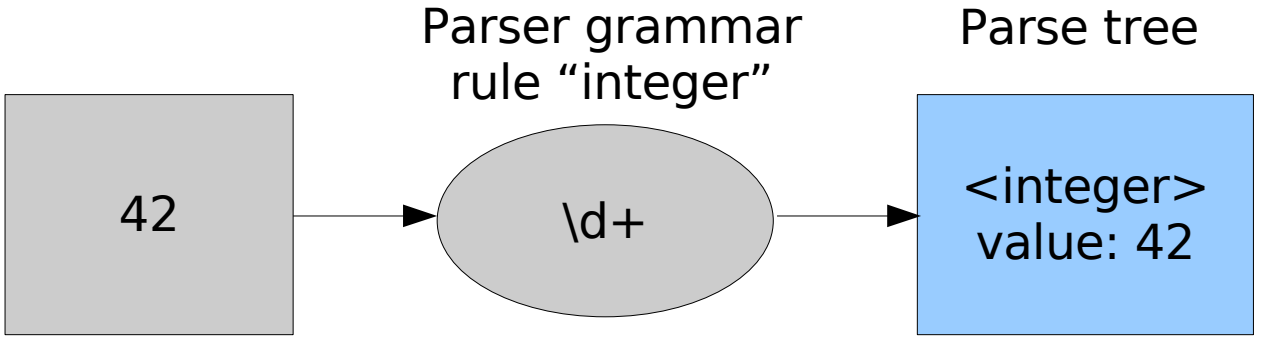


Value Transformation

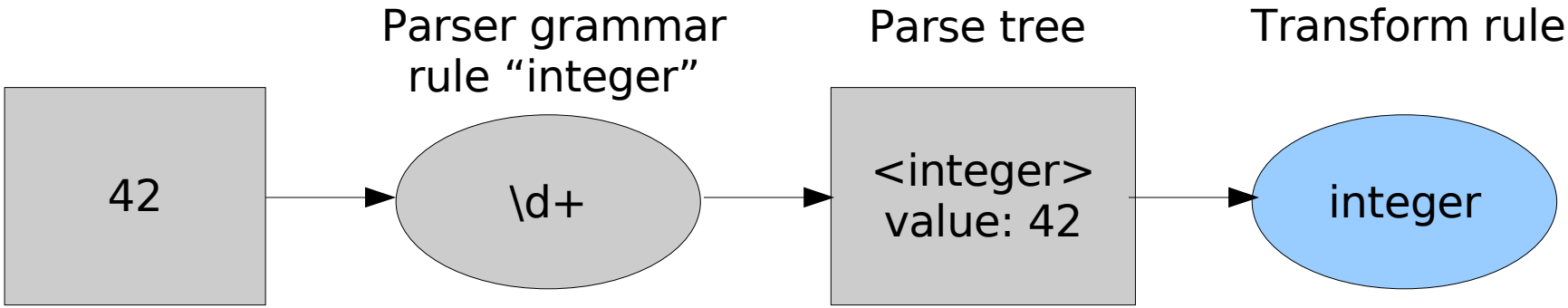


token integer { \d+ }

Value Transformation

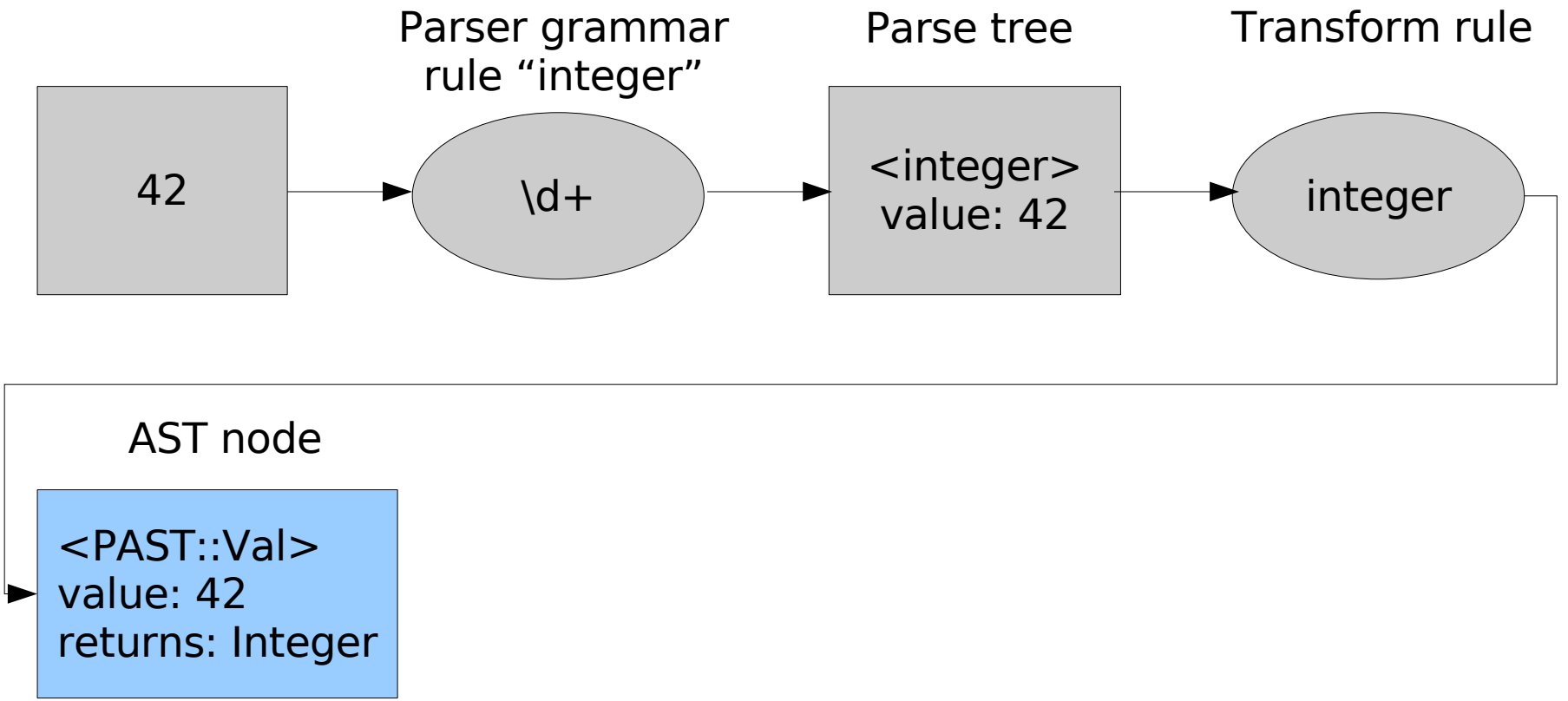


Value Transformation

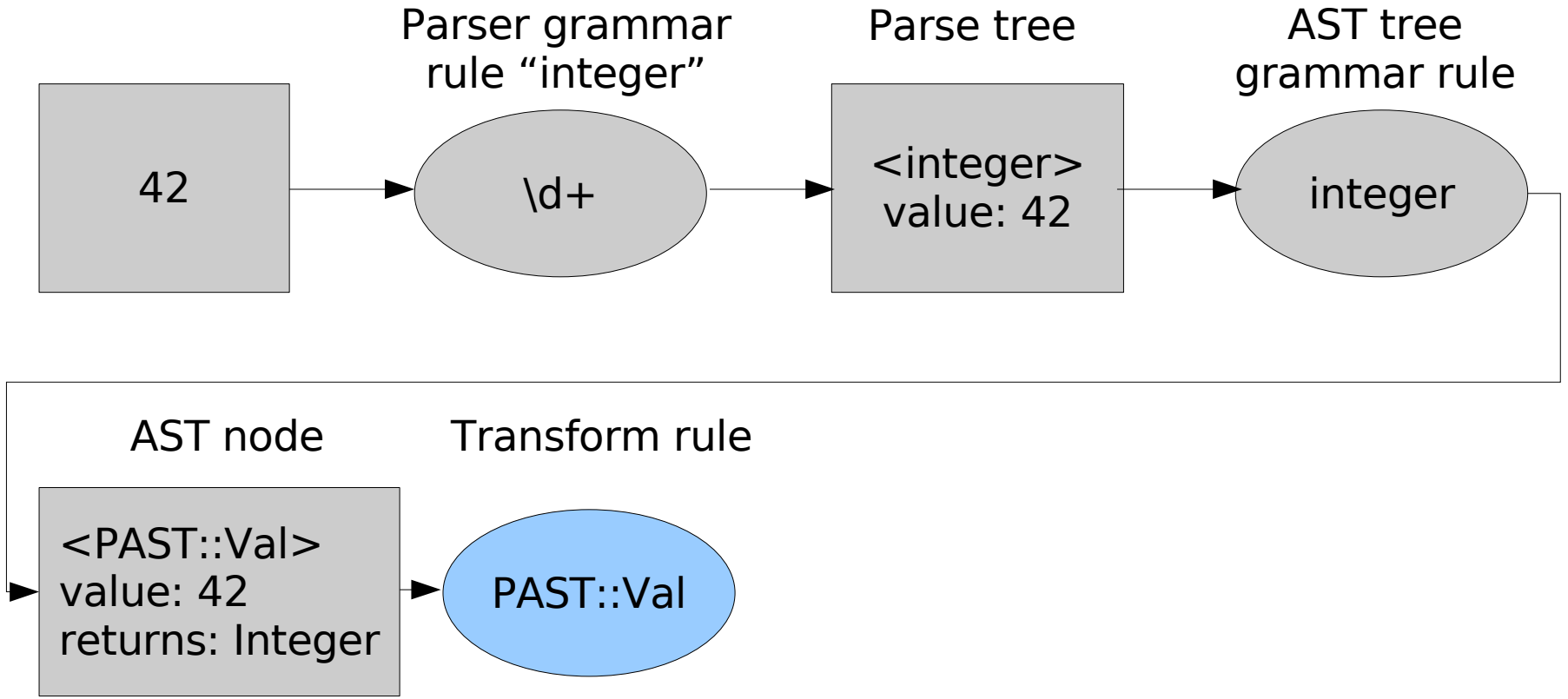


method integer($/) {...}

Value Transformation

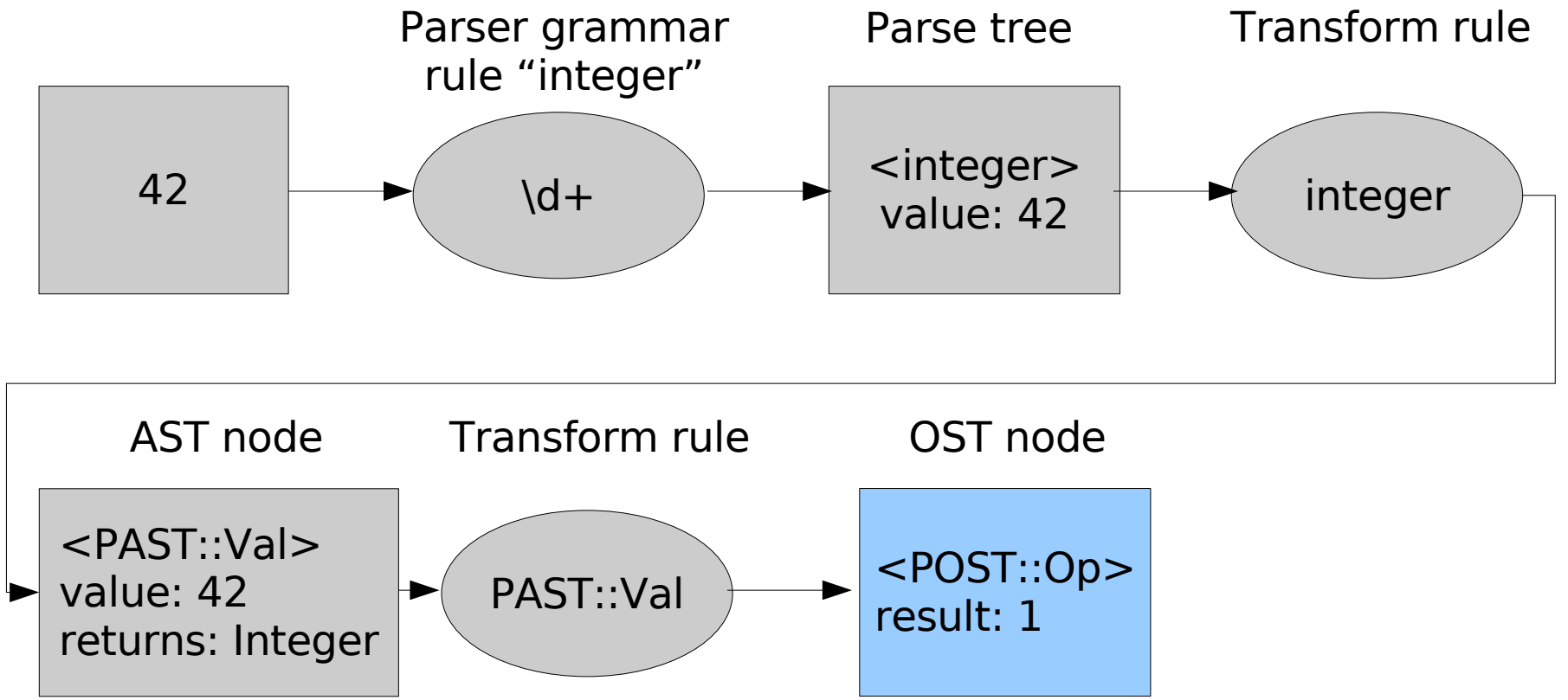


Value Transformation

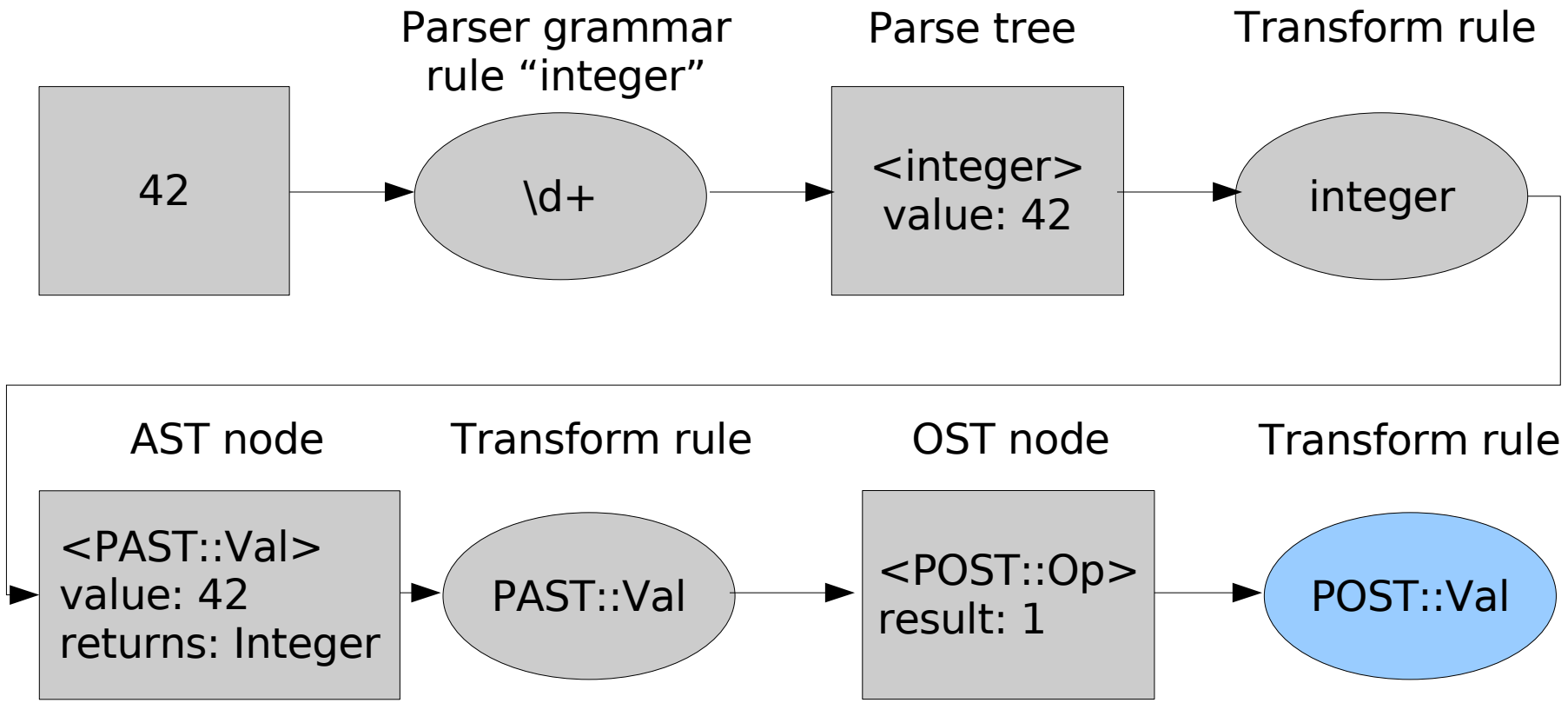


ost = self.as_post(ast)

Value Transformation

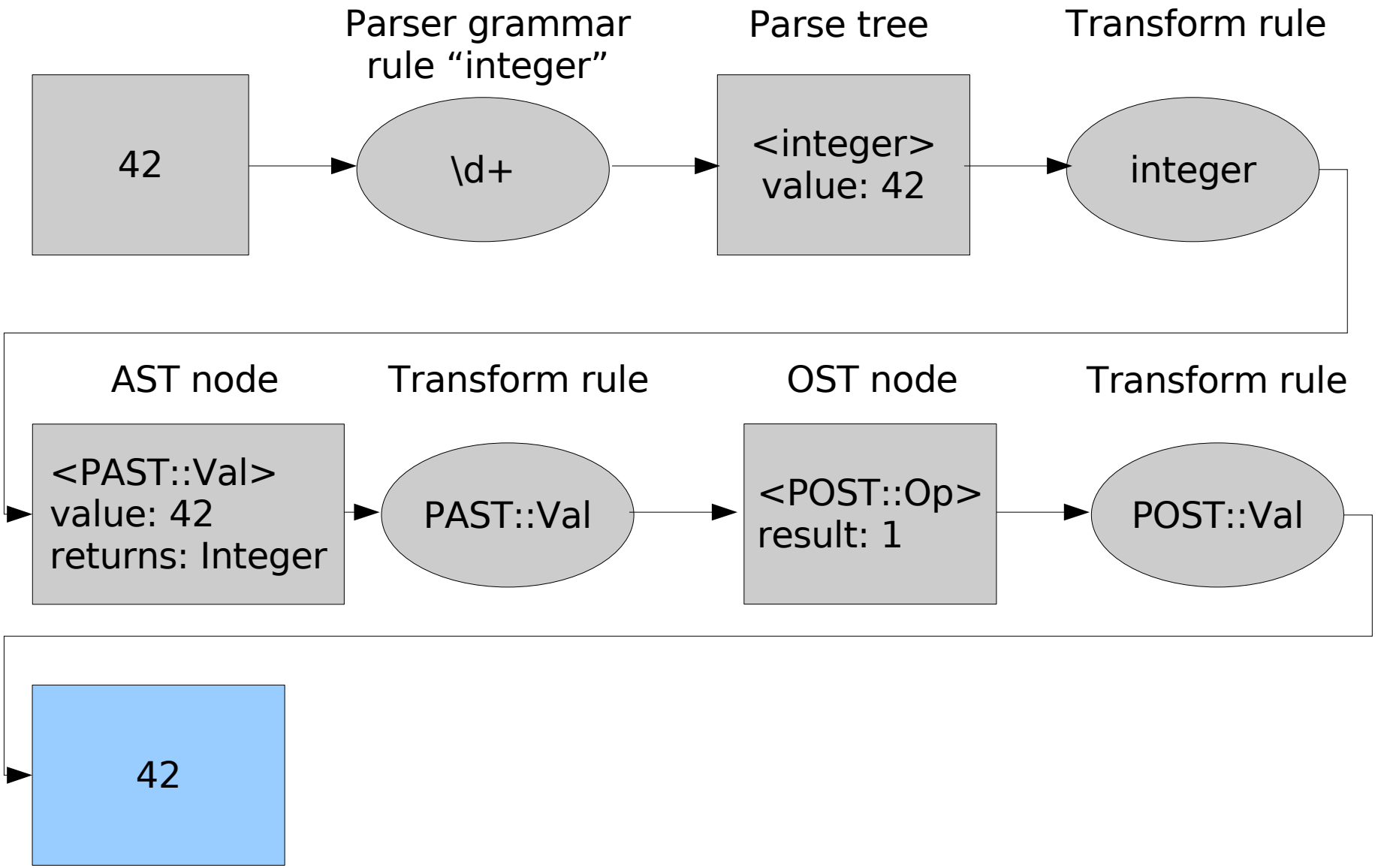


Value Transformation

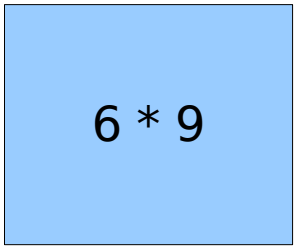


self.pir(ost)

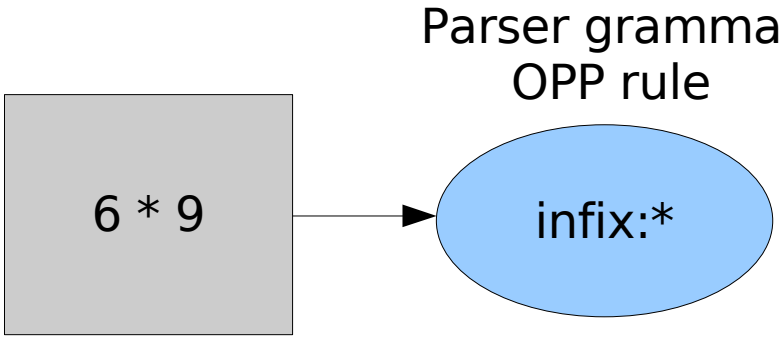
Value Transformation



Operator Transformation

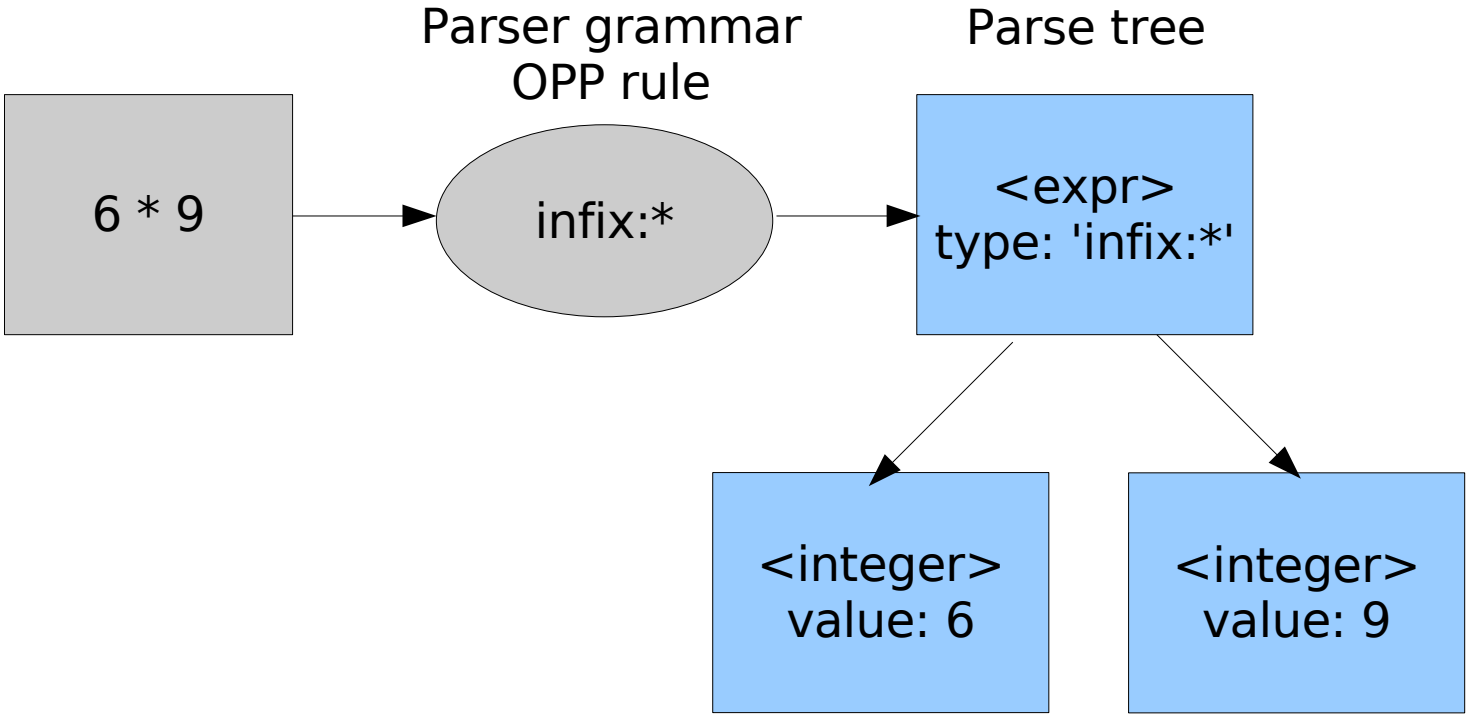


Operator Transformation

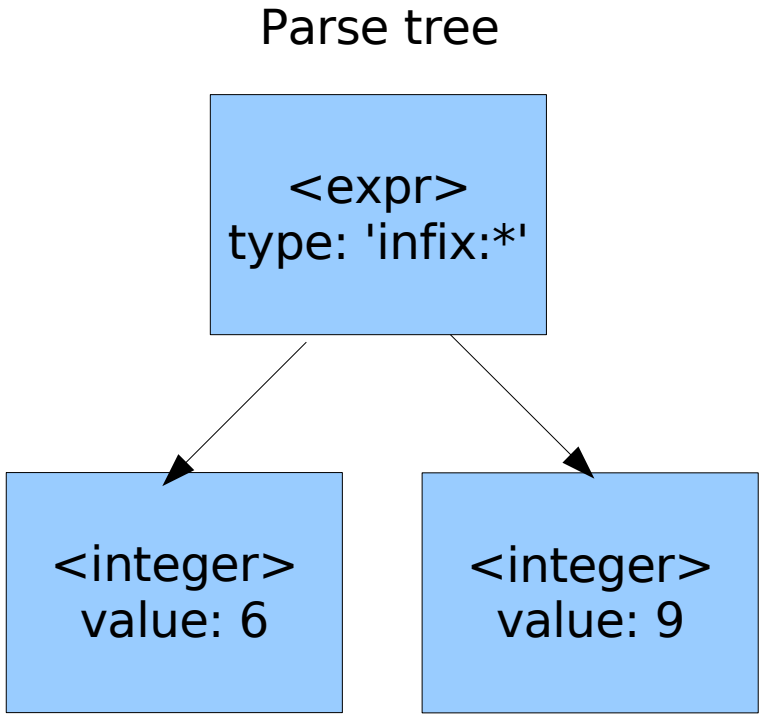


proto infix:* is looser(prefix:+) {...}

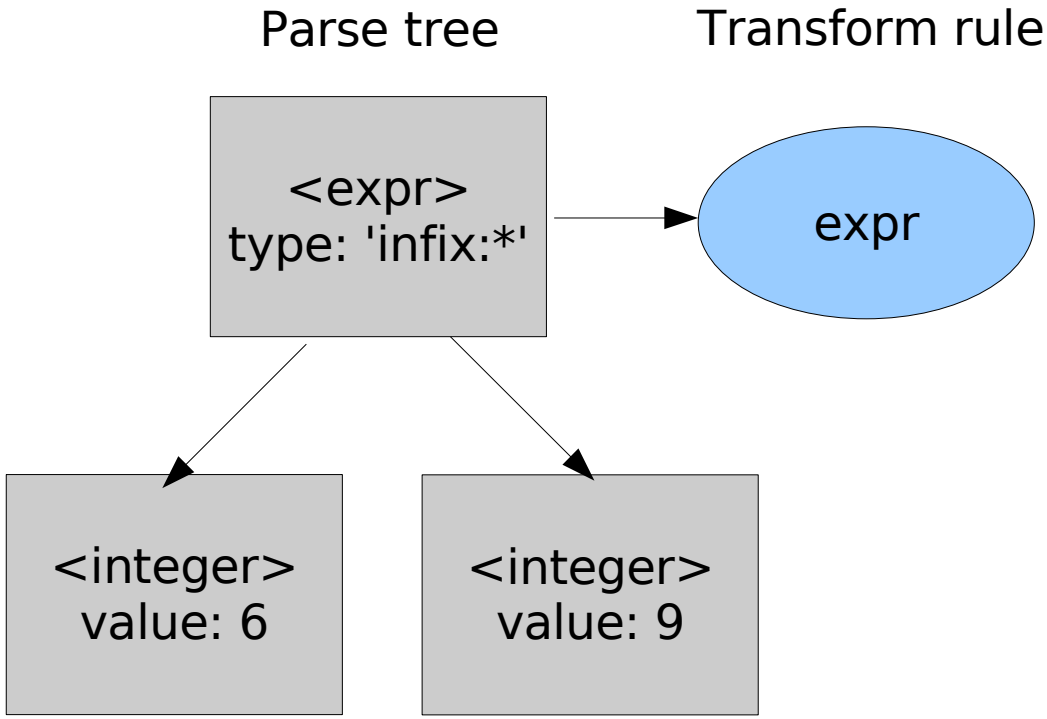
Operator Transformation



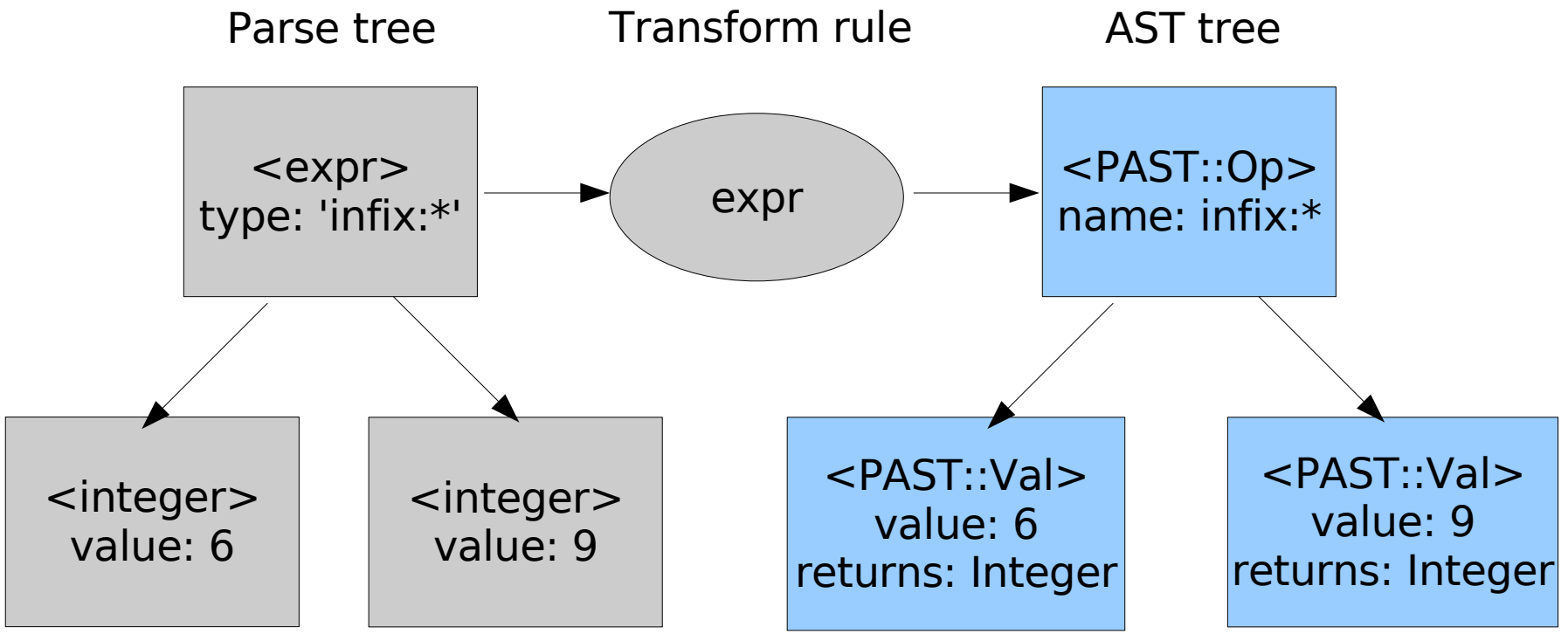
Operator Transformation



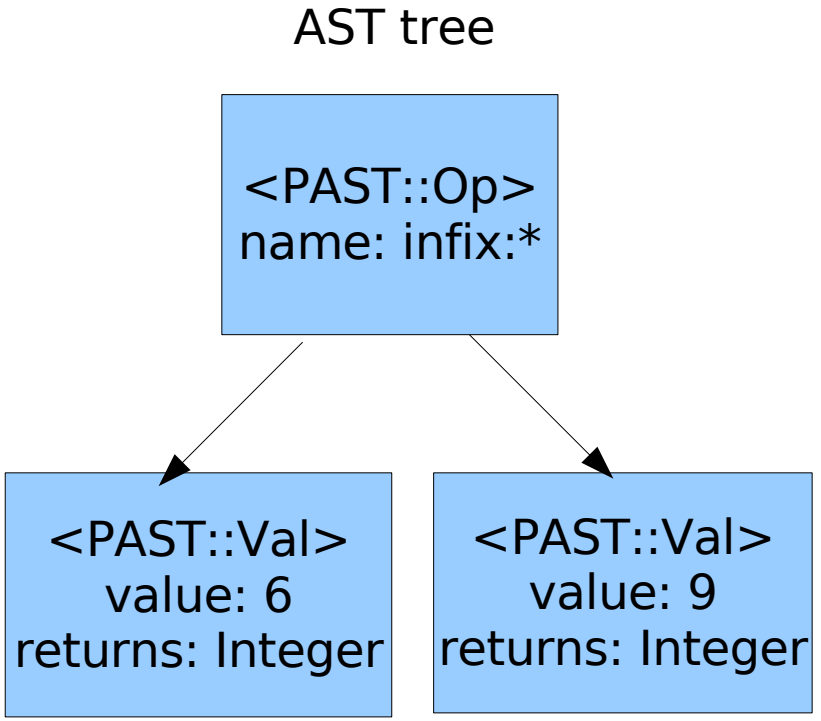
Operator Transformation



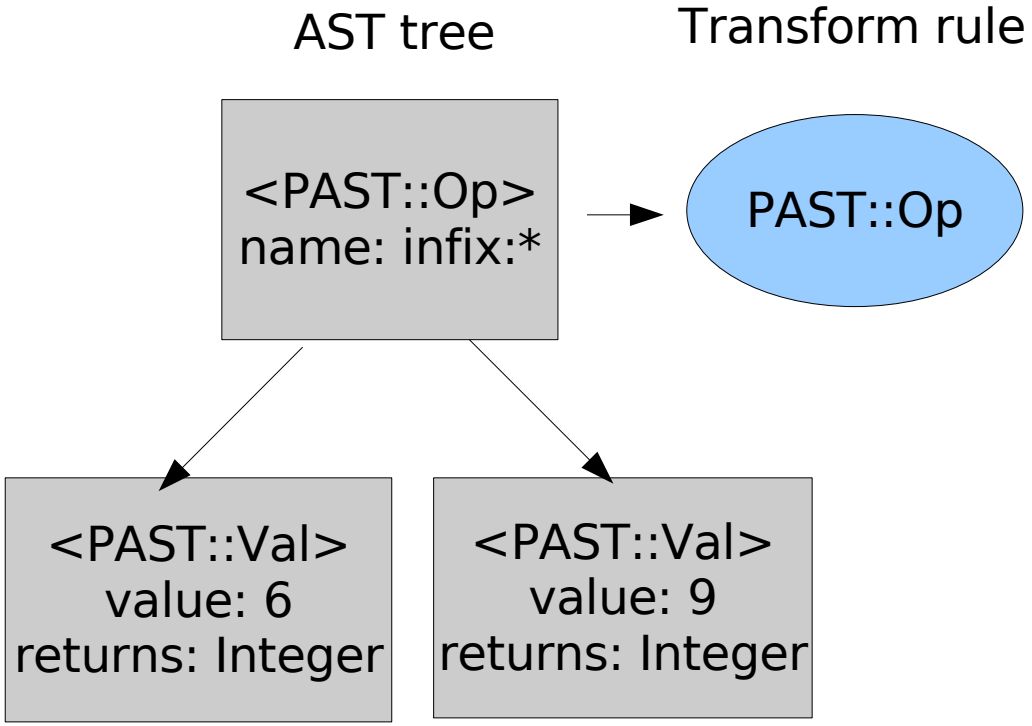
Operator Transformation



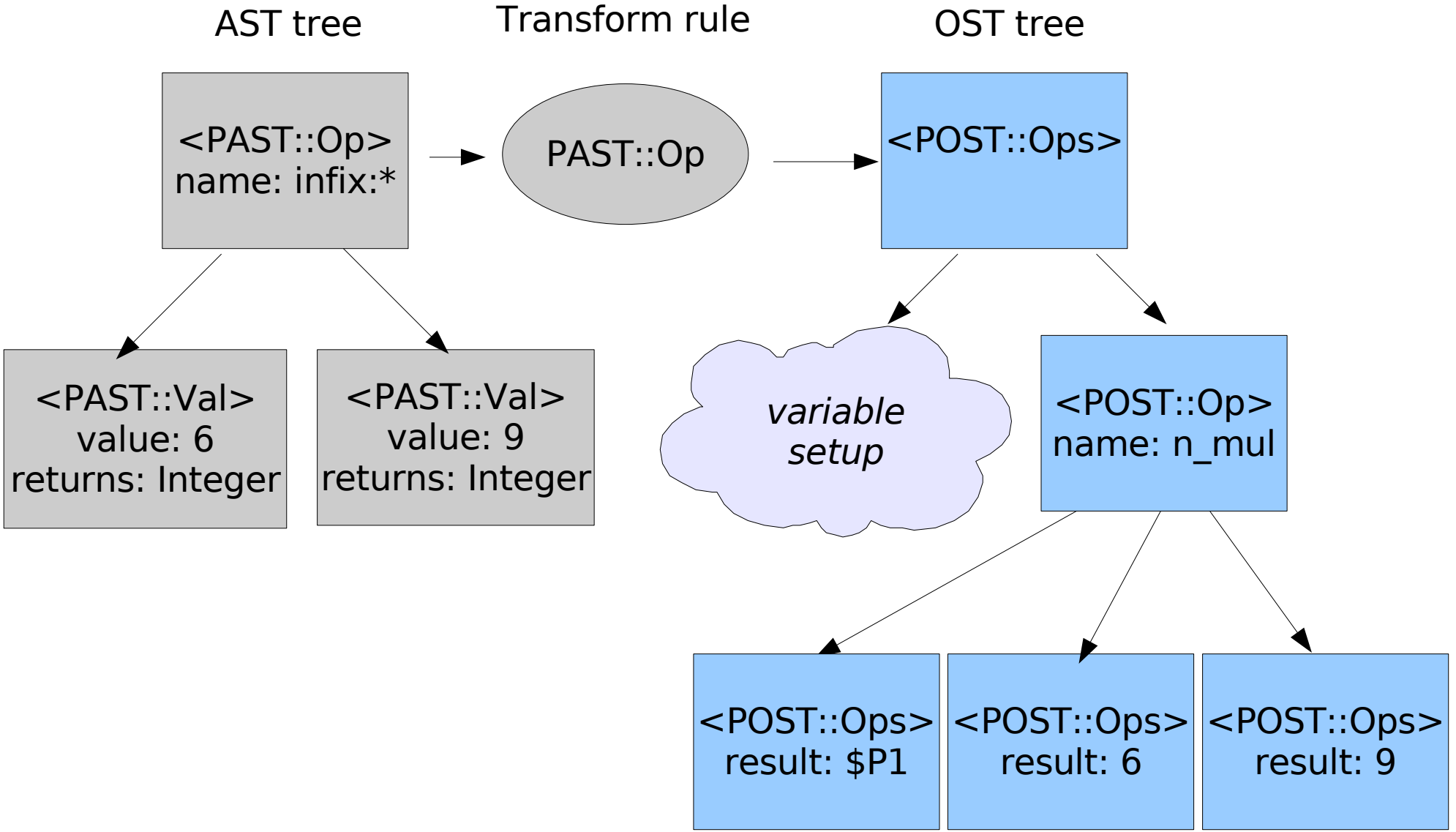
Operator Transformation



Operator Transformation



Operator Transformation



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
```

Questions?

- Further Reading
  - “Continuations and advanced flow control” by Jonathan Bartlett *<http://www.ibm.com/developerworks/linux/library/l-advflow.html>*
  - “The case for virtual register machines” by Brian Davis, et al. *<http://portal.acm.org/citation.cfm?id=858575>*