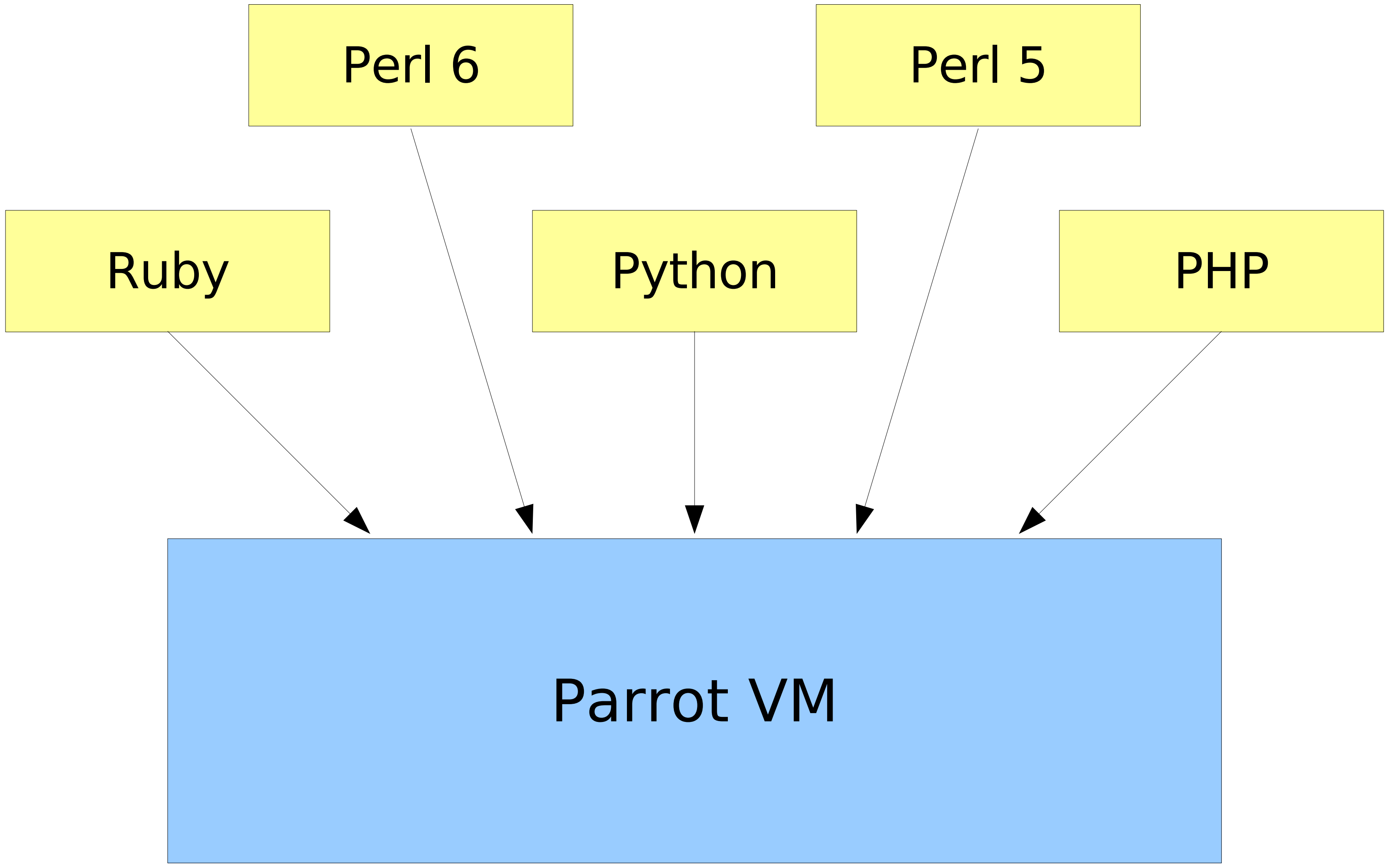
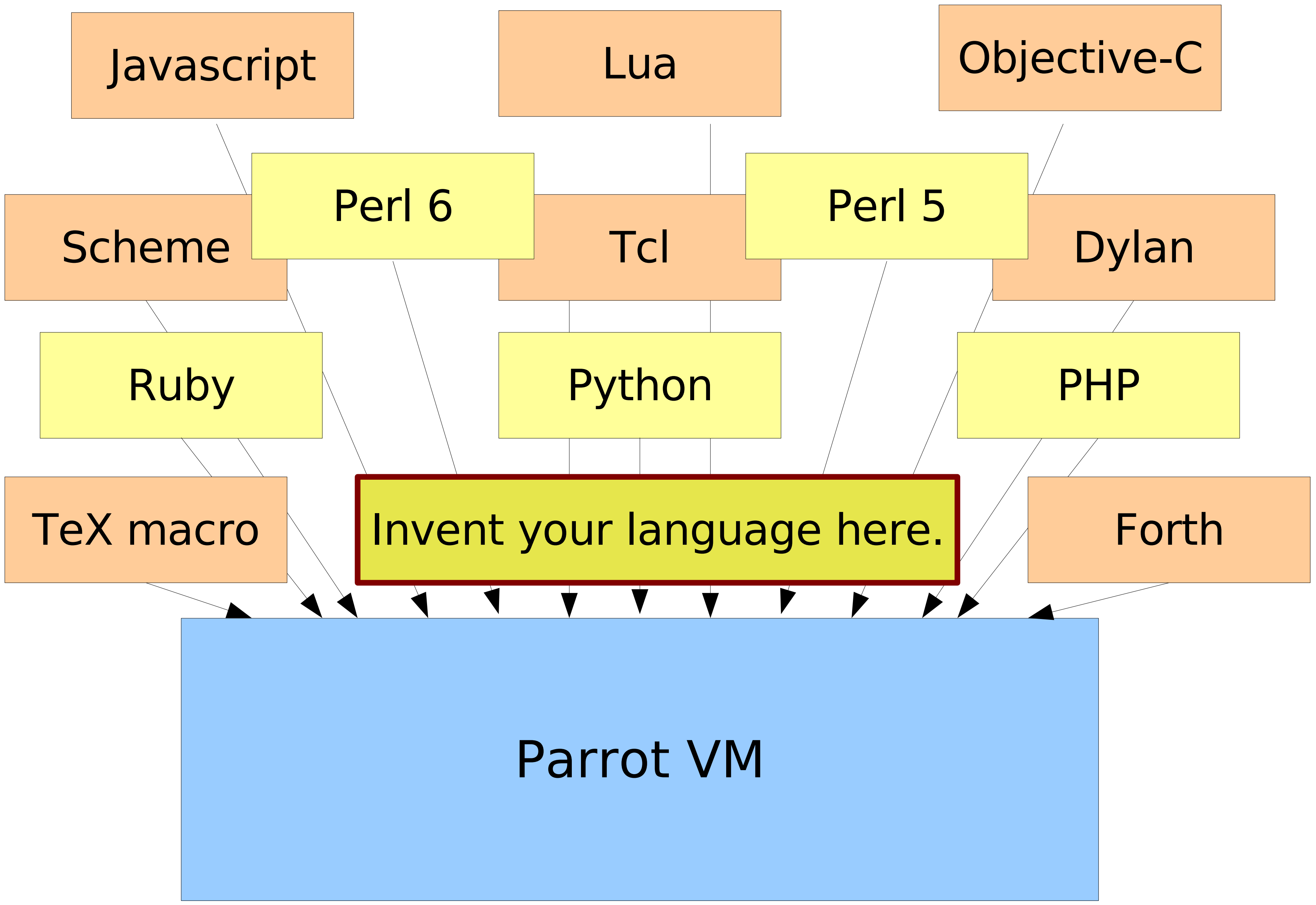
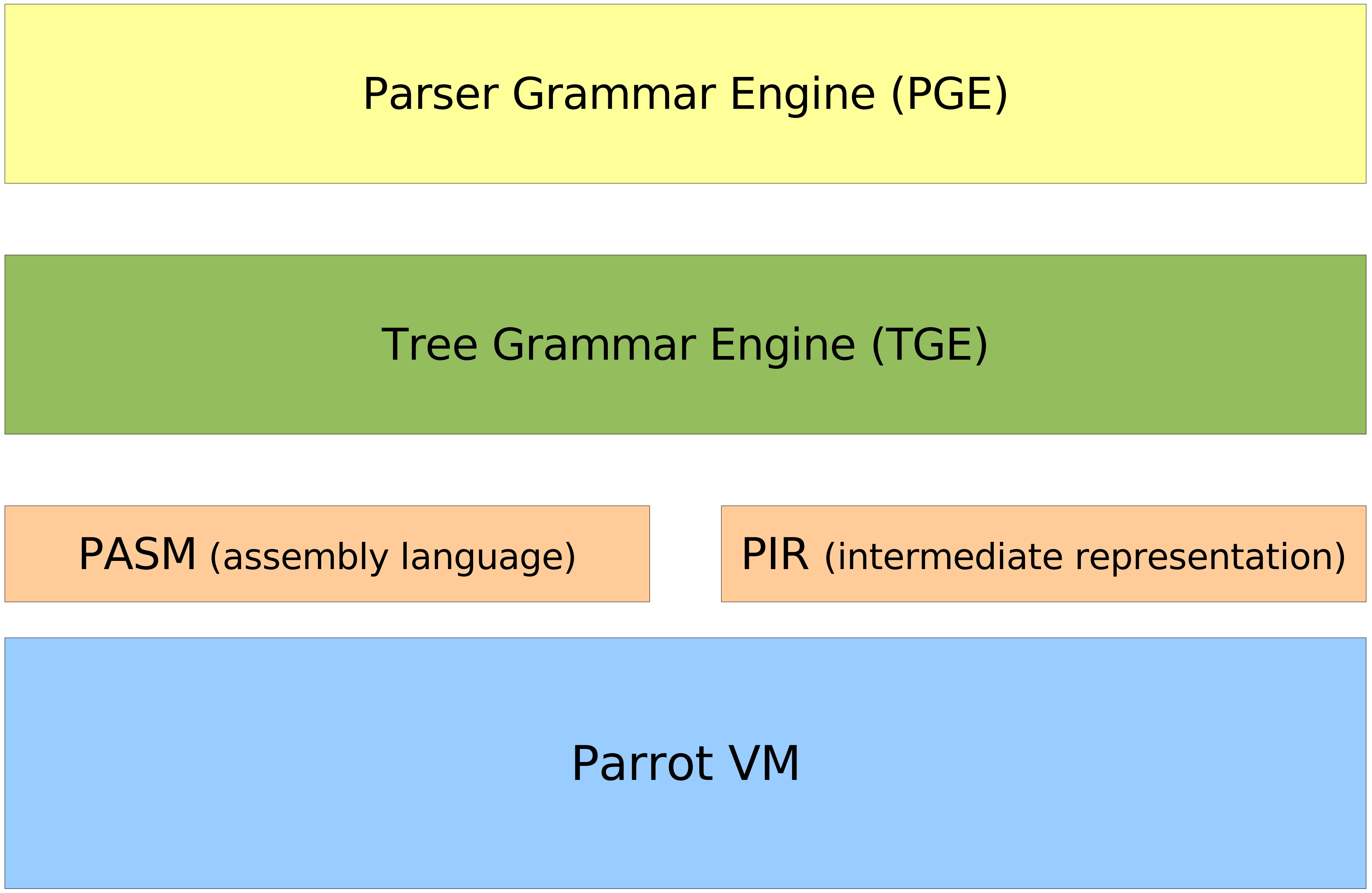
Parrot Compiler Tools

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.


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

Parser Grammar Engine

- Regular expressions
- Recursive descent
- Operator precedence parser

Parser Grammar Engine

- Parsing is recognizing patterns

```
if                    # "if" keyword
  a == 4              # an expression
then                  # "then" keyword
  print "Hello";      # a statement
```

- Grammar rules are patterns

```
rule conditional {
    if   <expression>
    then <statement>
}
```

Parser Grammar Engine

- PGE is a pattern compiler

```
grammar 'Simple';
rule ident { [ <alpha> | _ ] \w* }
```

- Run `pgc.pir`

```
$ parrot pgc.pir simple.pg > simple.pir
```

- PIR output

```
.sub 'ident'
    ... # 308 lines
.end
```

Parser Grammar Engine

- Use the compiled parser

```
.sub 'foo'
  ...
  load_bytecode 'simple.pir'
  # retrieve the rule sub
  parse = find_global 'Simple', 'ident'

  source = '_identifier' # the source
  match = parse(source)  # parse

  '_dumper'(match) # dump the tree
.end
```

Parser Grammar Engine

- Rule, token, regex
- Rule and token don't backtrack
- Rule does smart whitespace matching

```
rule conditional {
  if   <expression>
  then <statement>
}
token conditional {
  \s* if   \s* <expression>
  \s* then \s* <statement> \s*
}
```

Parser Grammar Engine

- Operator precedence parser

```
proto infix:+ is precedence('=') { ... }
proto infix:- is equiv('infix:+') { ... }

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

- Associativity

```
proto infix:= is assoc('right') is
  looser('infix:||') { ... }
```

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
- Transforming trees

Tree Grammar Engine

- TGE is a transform compiler

```
grammar ASTGen is TGE::Grammar;

transform astout (ident) :language('PIR') {
    .local pmc result
    result = new 'AST::Ident'
    $S2 = node
    result.'name'($S2)
    ...
    .return (result)
}
```

Tree Grammar Engine

- Run `tgc.pir`

```
$ parrot tgc.pir ASTGen.tg > ASTGen.pir
```

- PIR output

```
.sub '_ident_astout' :method
    .param pmc tree
    .param pmc node
    ...
.end
```

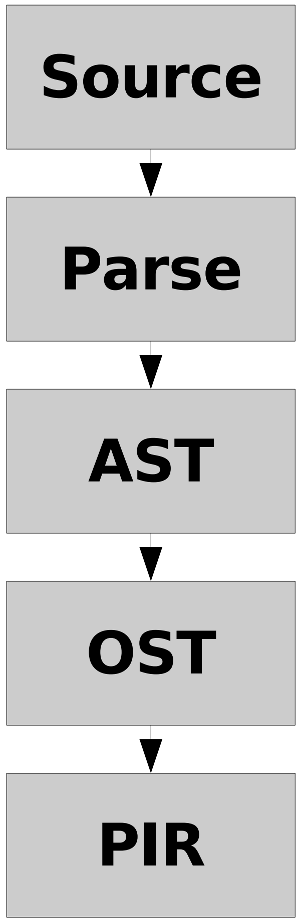
Tree Grammar Engine

- Use the compiled transformer

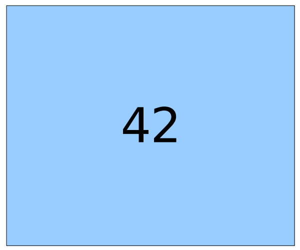
```
.sub 'foo'
  load_bytecode 'ASTGen.pir'
  ...
  grammar = new 'ASTGen'
  astbuilder = grammar.apply(matchtree)
  ast = astbuilder.get('astout')

  ast.dump()
.end
```

Compiler Tools

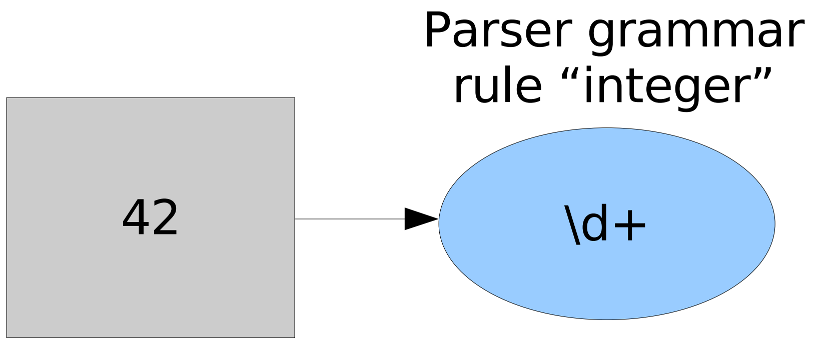
- 4 stages
- Parse Tree
- Abstract Syntax Tree
- Opcode Syntax Tree
- PIR (or bytecode)



Value Transformation

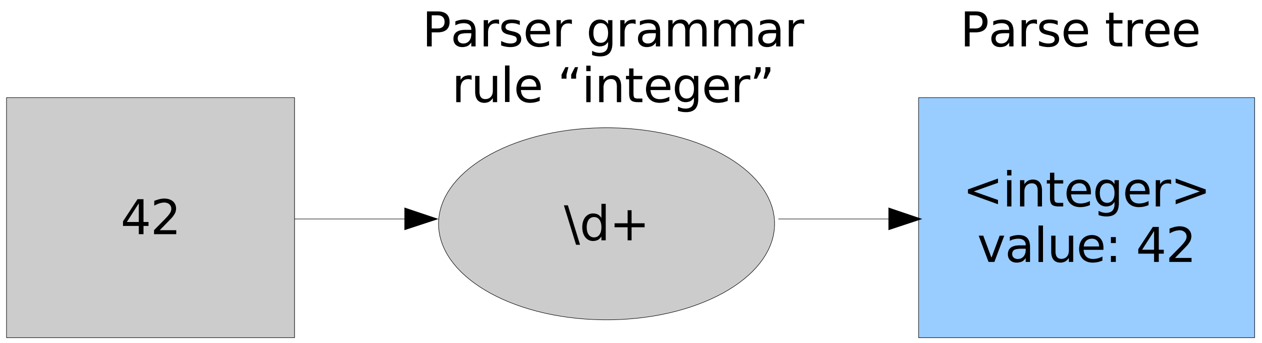


Value Transformation

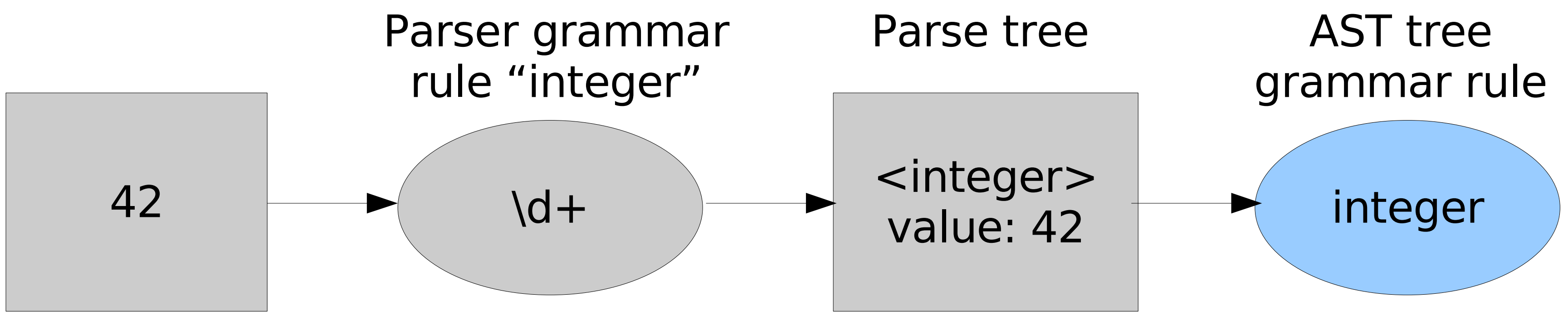


```
token integer { \d+ }
```

Value Transformation

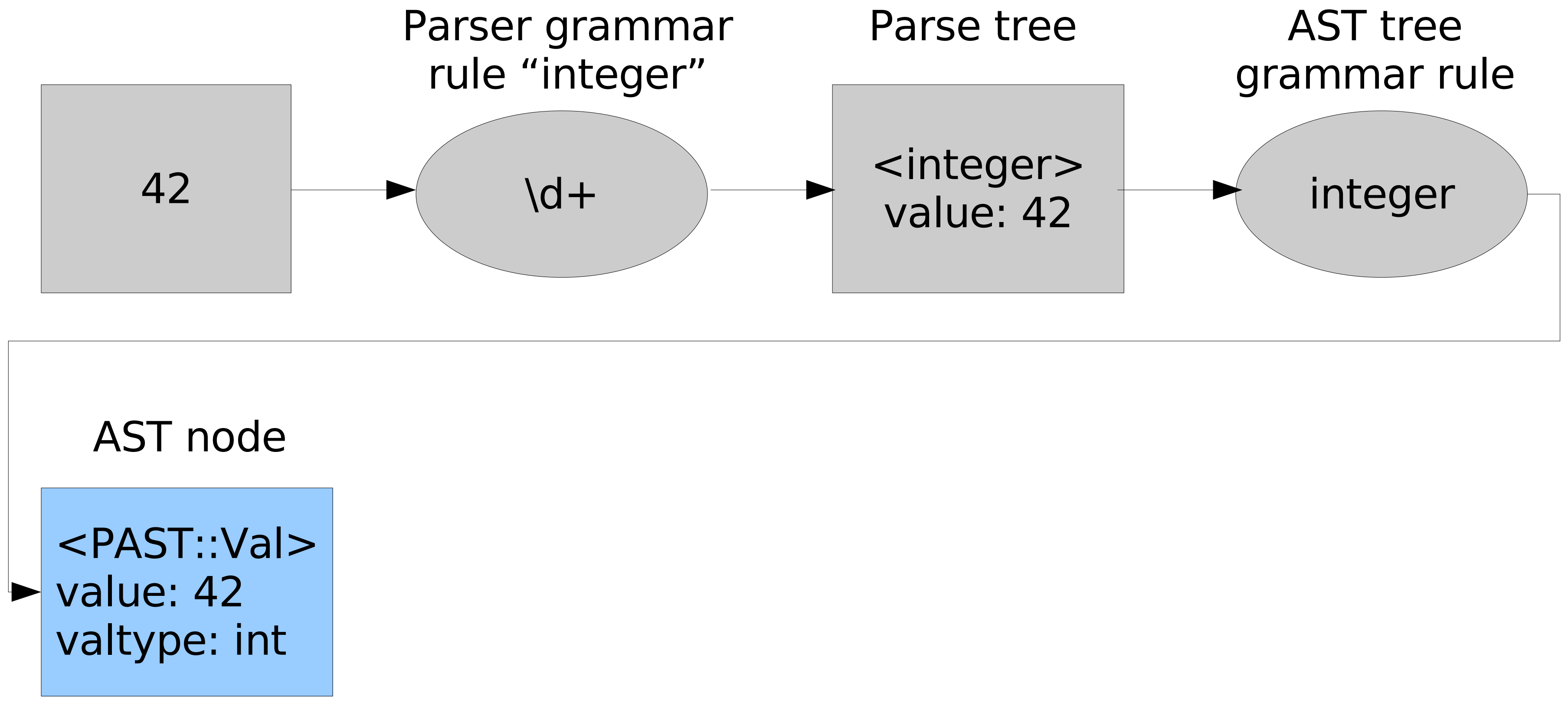


Value Transformation

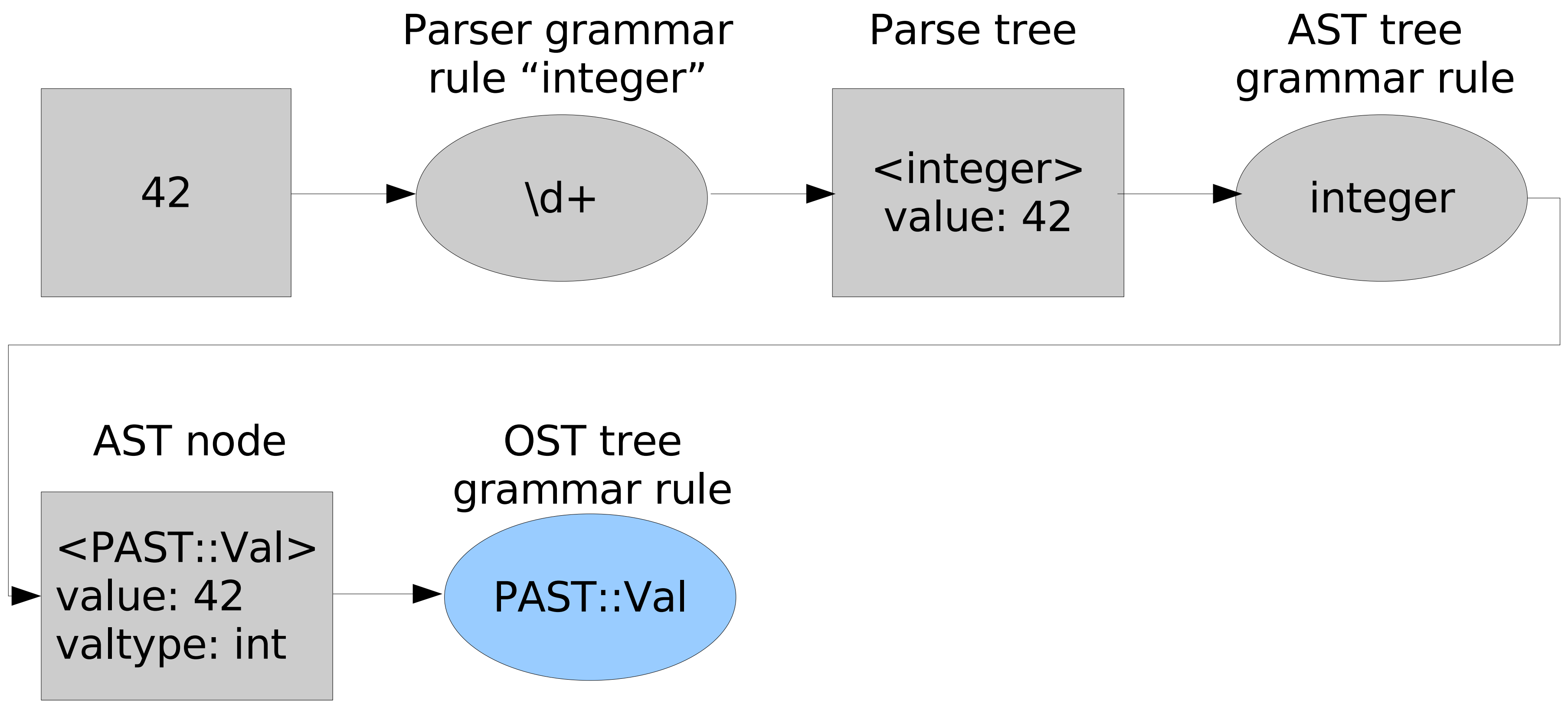


```
transform buildast (integer) {...}
```

Value Transformation

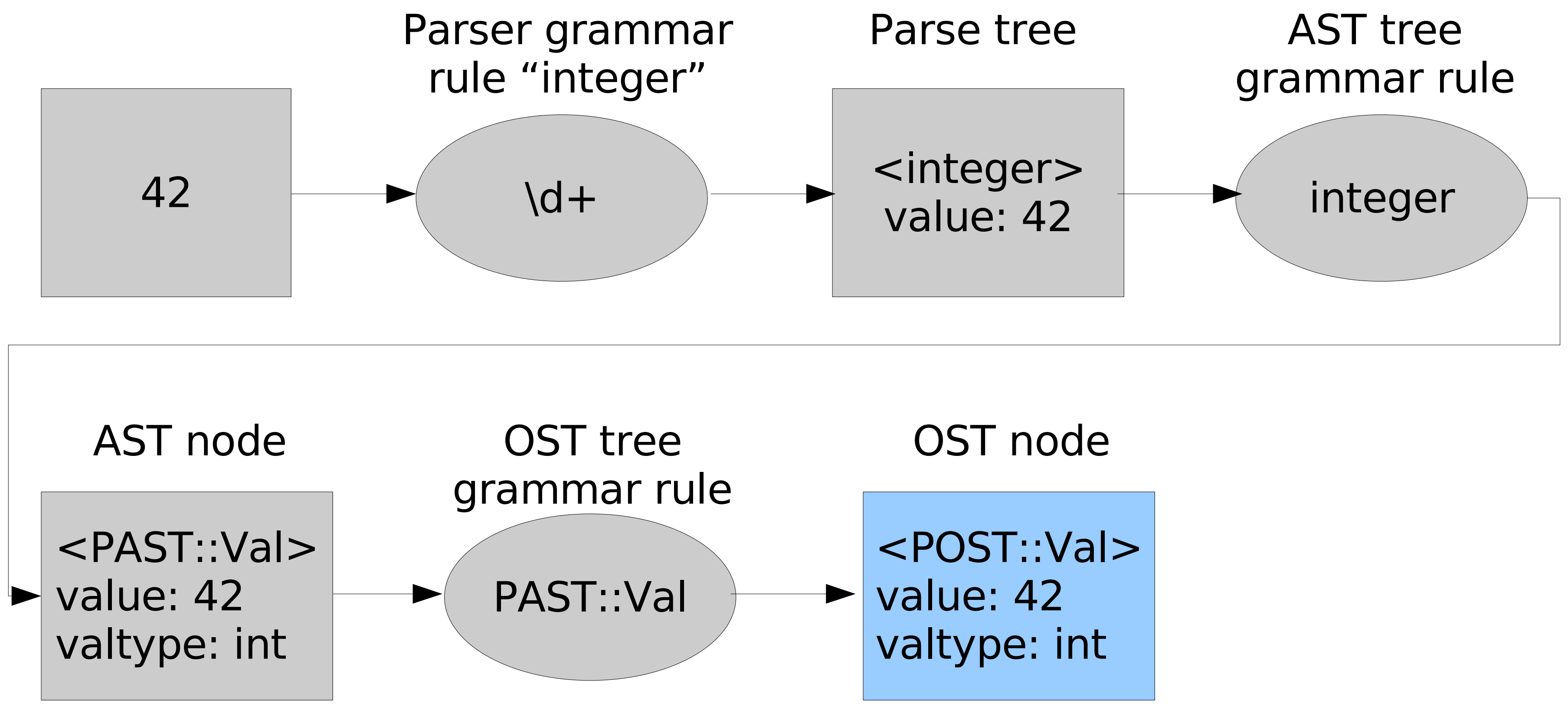


Value Transformation

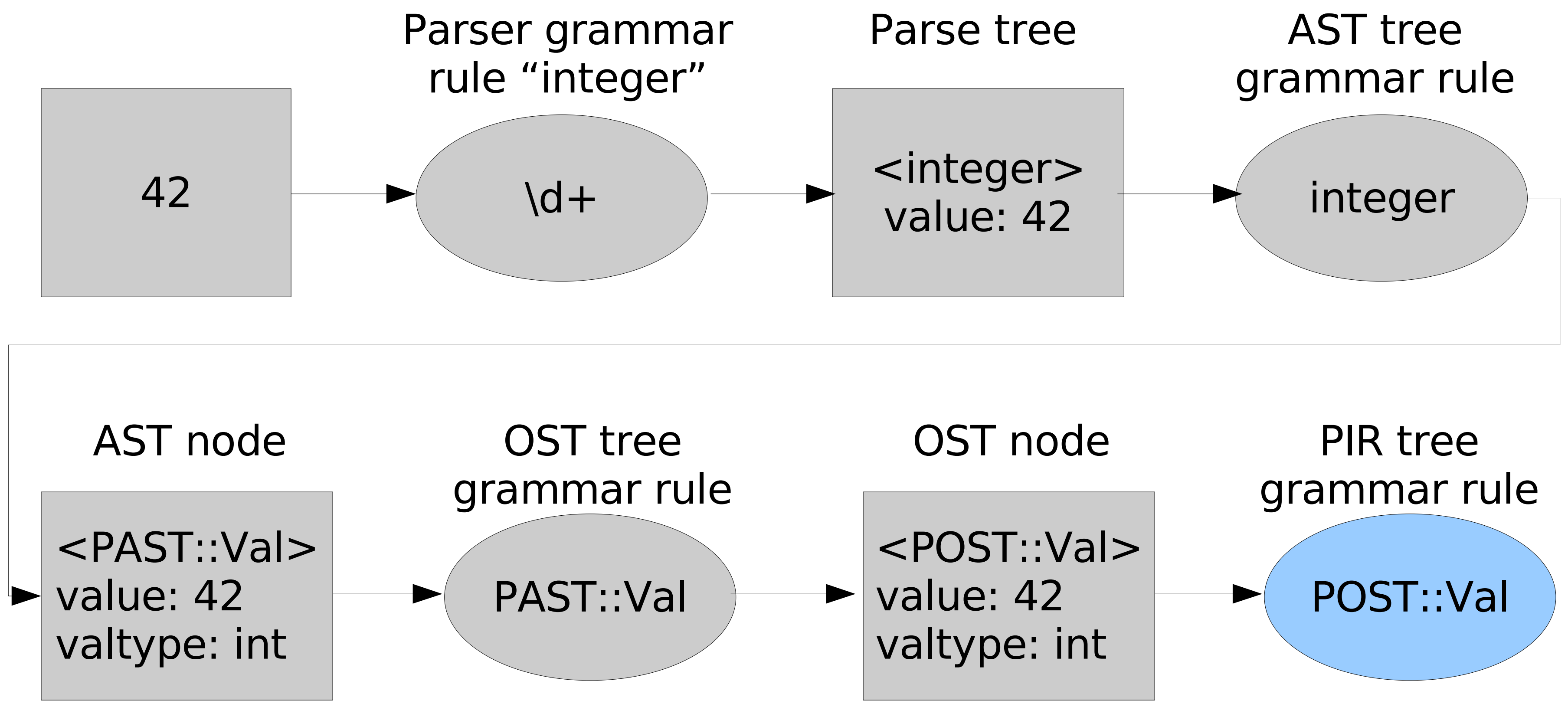


transform buildost (PAST::Val) {...}

Value Transformation

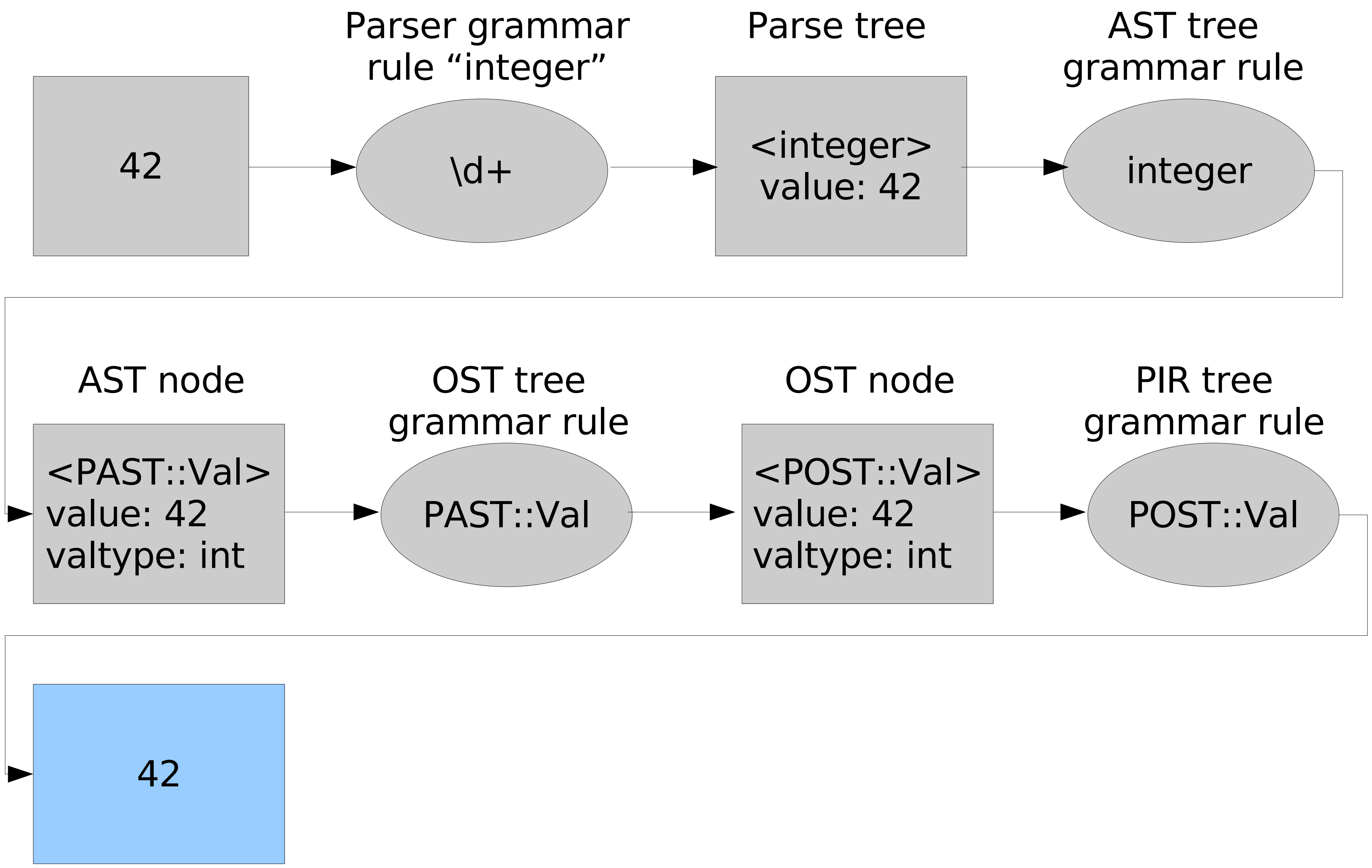


Value Transformation



transform buildpir (POST::Val) {...}

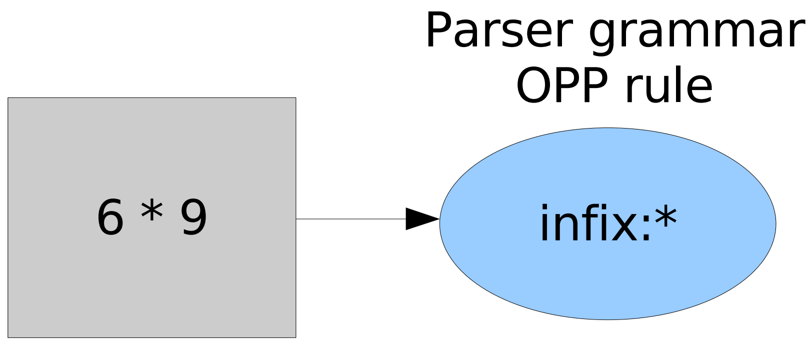
Value Transformation



Operator Transformation

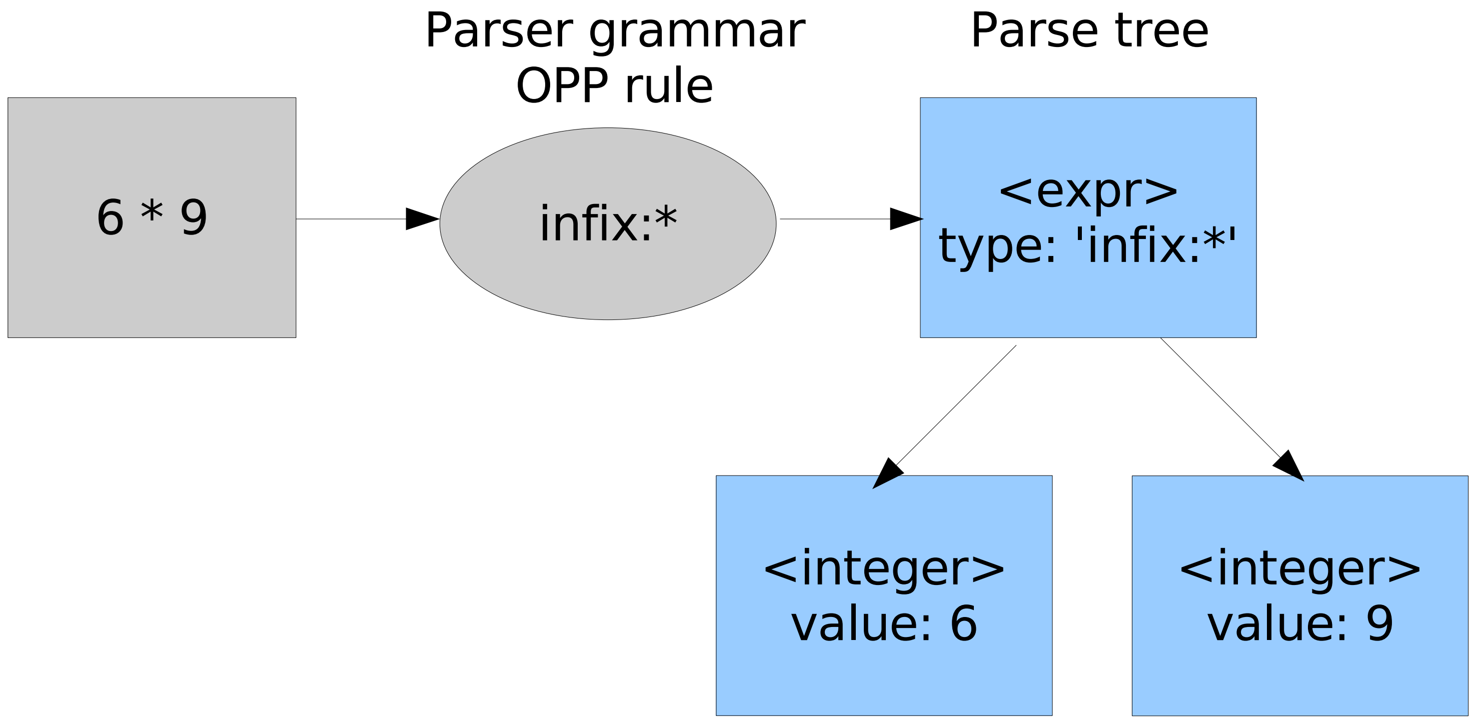
6 * 9

Operator Transformation



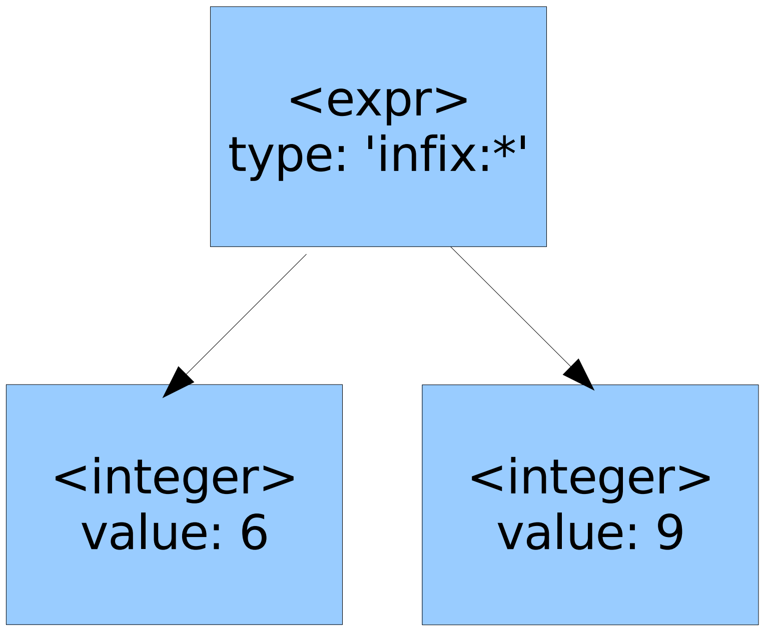
proto 'infix:*' is tighter('infix:+') {...}

Operator Transformation

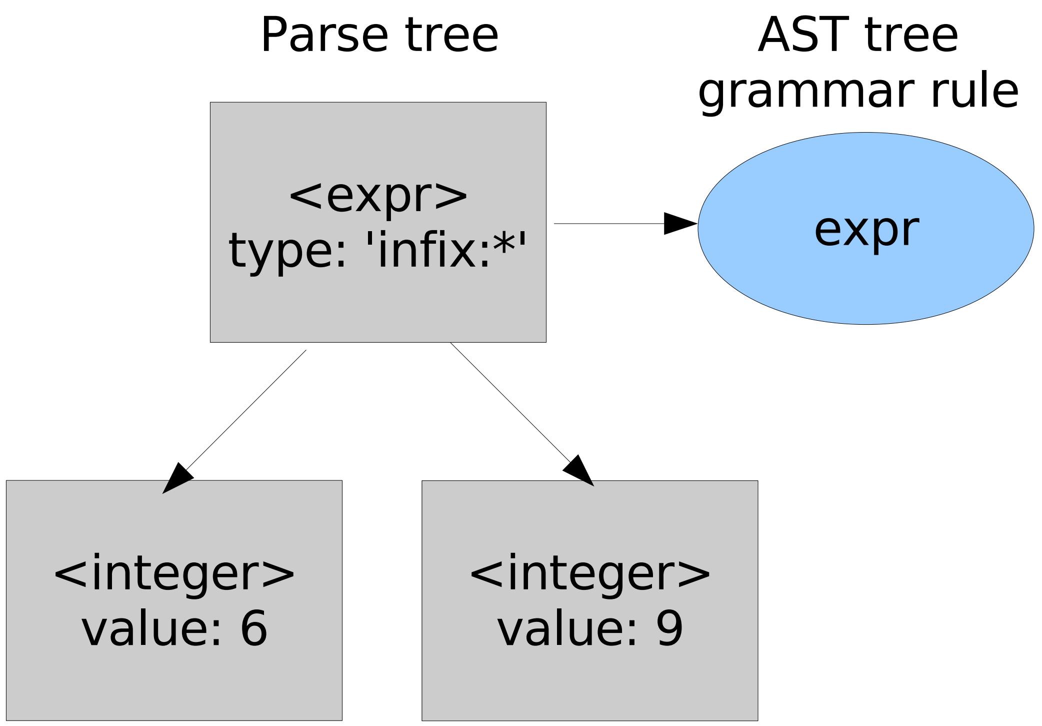


Operator Transformation

Parse tree

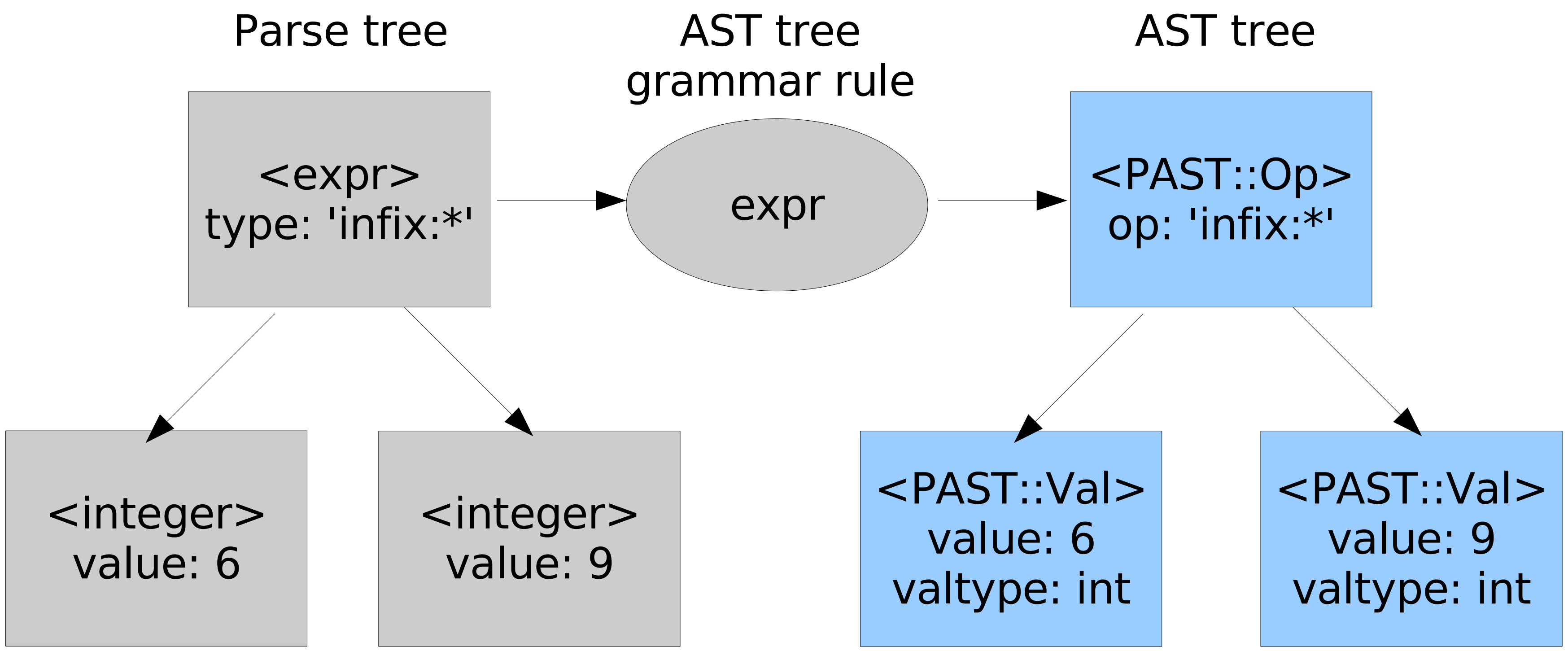


Operator Transformation



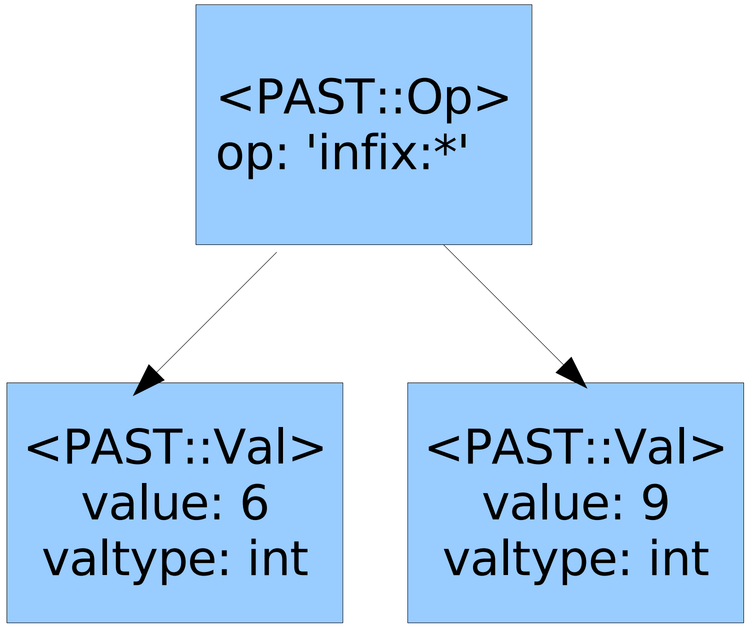
transform buildast (expr) {...}

Operator Transformation

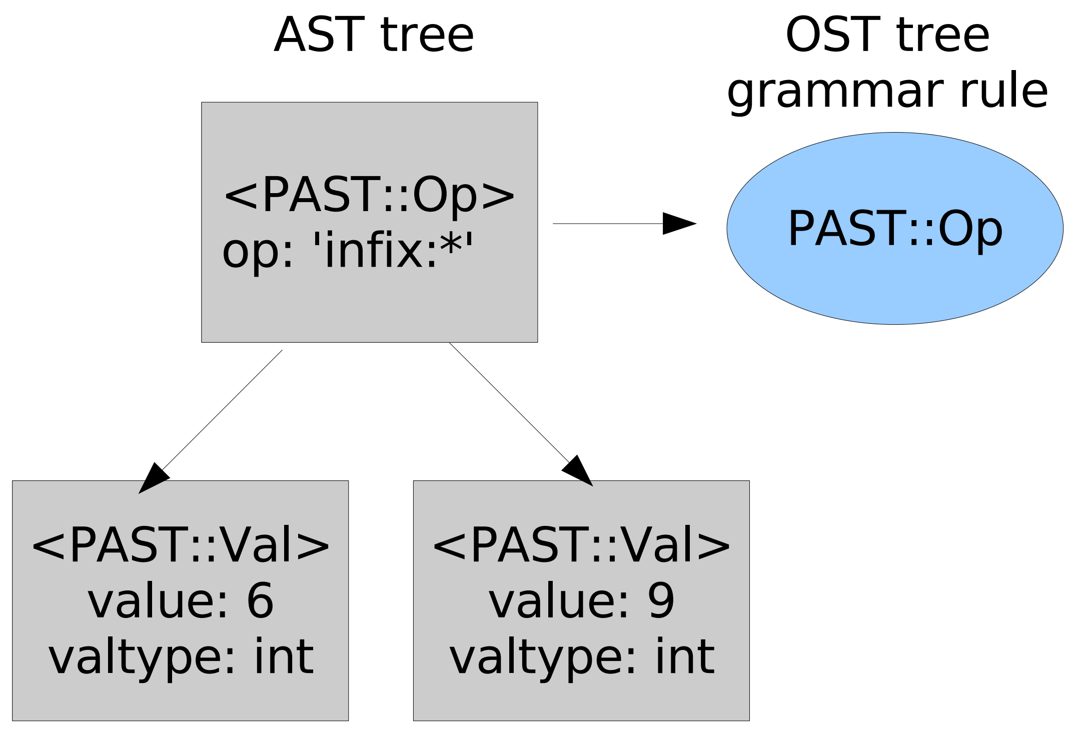


Operator Transformation

AST tree

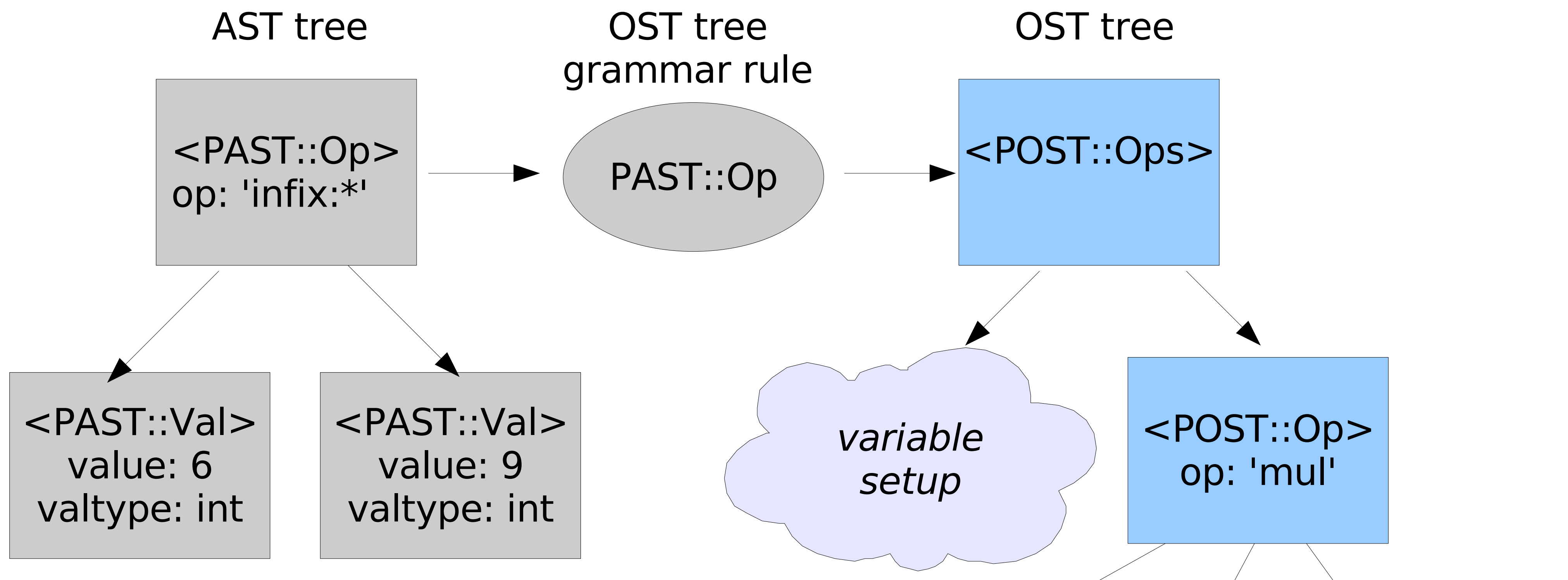


Operator Transformation


AST tree
OST tree grammar rule
<PAST::Op> op: 'infix:*'
PAST::Op
<PAST::Val> value: 6 valtype: int
<PAST::Val> value: 9 valtype: int

transform buildost (PAST::Op) {...}

Operator Transformation



Operator Transformation

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

Summary

- Attract multiple languages
- Easy to use
- Simple steps
- Hide Complexity
- ~~Im~~possible

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.