Formal Theory, Informally

Jonathan Worthington

German Perl Workshop 2007

RED TAPE
HOLDS UP
NEW BRIDGE

INCLUDE YOUR CHILDREN WHEN BAKING COOKIES

PORTABLE TOILET BOMBED, POLICE HAVE NOTHING TO GO ON

Formal vs. Informal Languages

- English, German, etc. are informal languages
  - There is no definition of the language that gives every valid sentence or rejects every invalid sentence
  - Massively distributed and uncontrolled evolution of the language

Formal vs. Informal Languages

- Perl, mathematical expressions, and any regular language are examples of formal languages
  - We have a finite formal definition that tells us every possible valid “sentence” in the language and what it means
  - For Perl, you can consider the implementation to be that definition

Meta-language

- Using one language to talk about another
- Yesterday I used English as a meta-language for Perl 6
  - An informal meta-language for a formal language
- This talk is mostly about formal meta-languages for programming languages.

The Journey Of A Program

The Journey Of A Program

1. The program is tokenised

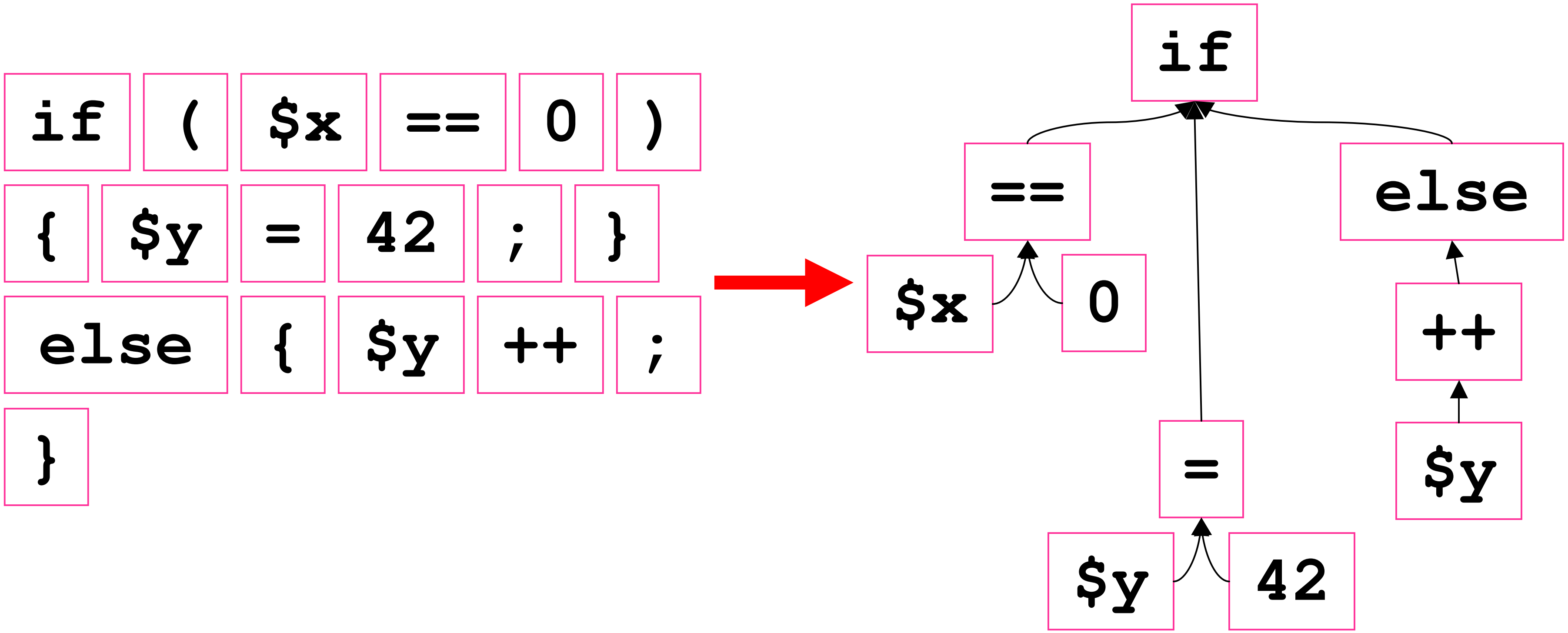
```
if ($x == 0) {
    $y = 42;
} else {
    $y++;
}
```

→

`if` `(` `$x` `==` `0` `)`
`{` `$y` `=` `42` `;` `}`
`else` `{` `$y` `++` `;`
`}`

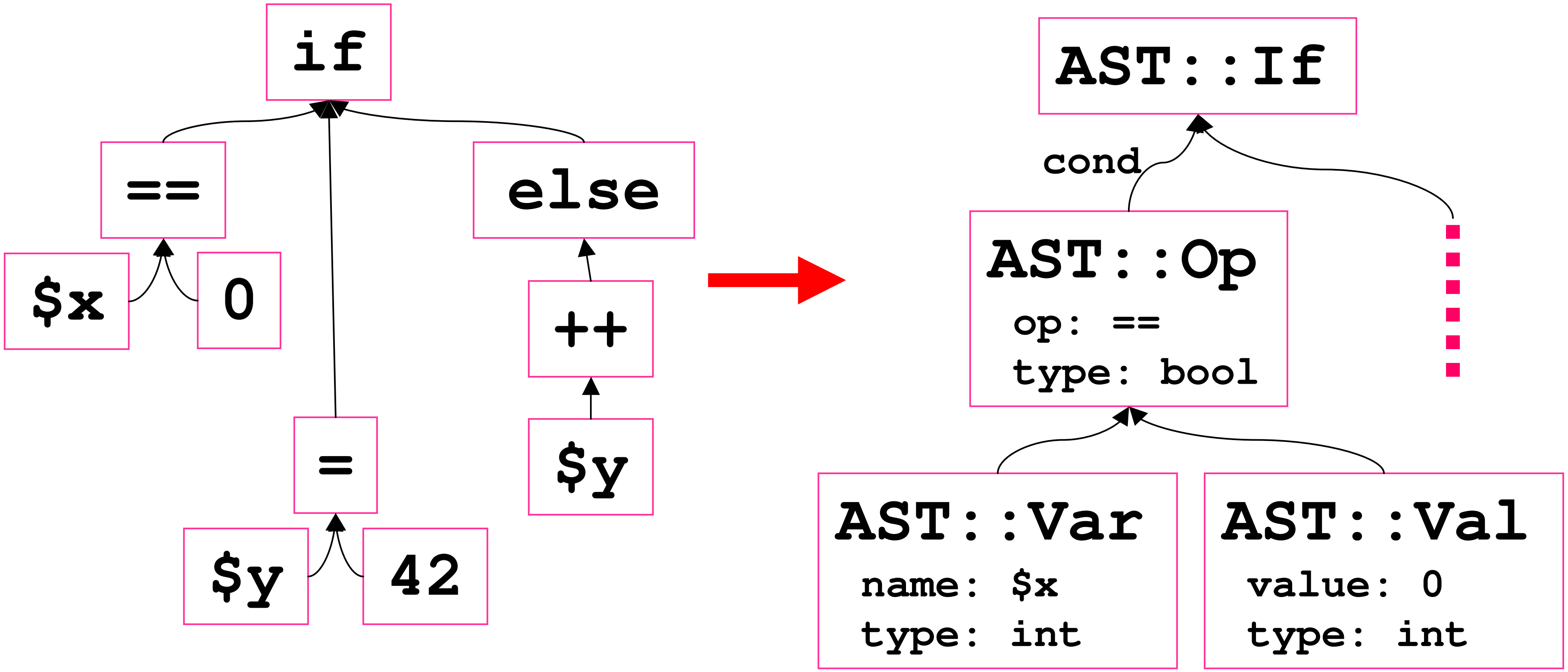
The Journey Of A Program

2. The parser takes these tokens and makes a parse tree



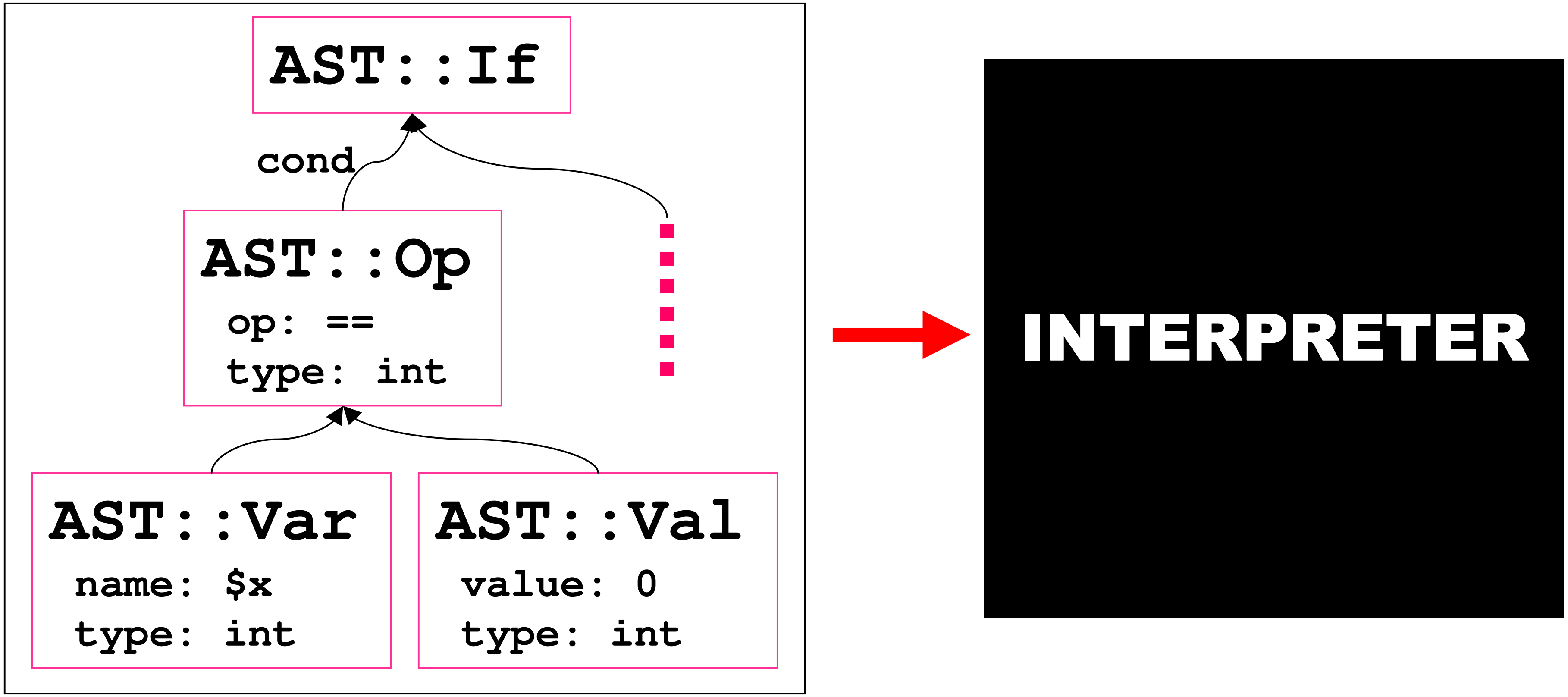
The Journey Of A Program

3. We do magical funky things to the tree and it becomes an abstract syntax tree



The Journey Of A Program

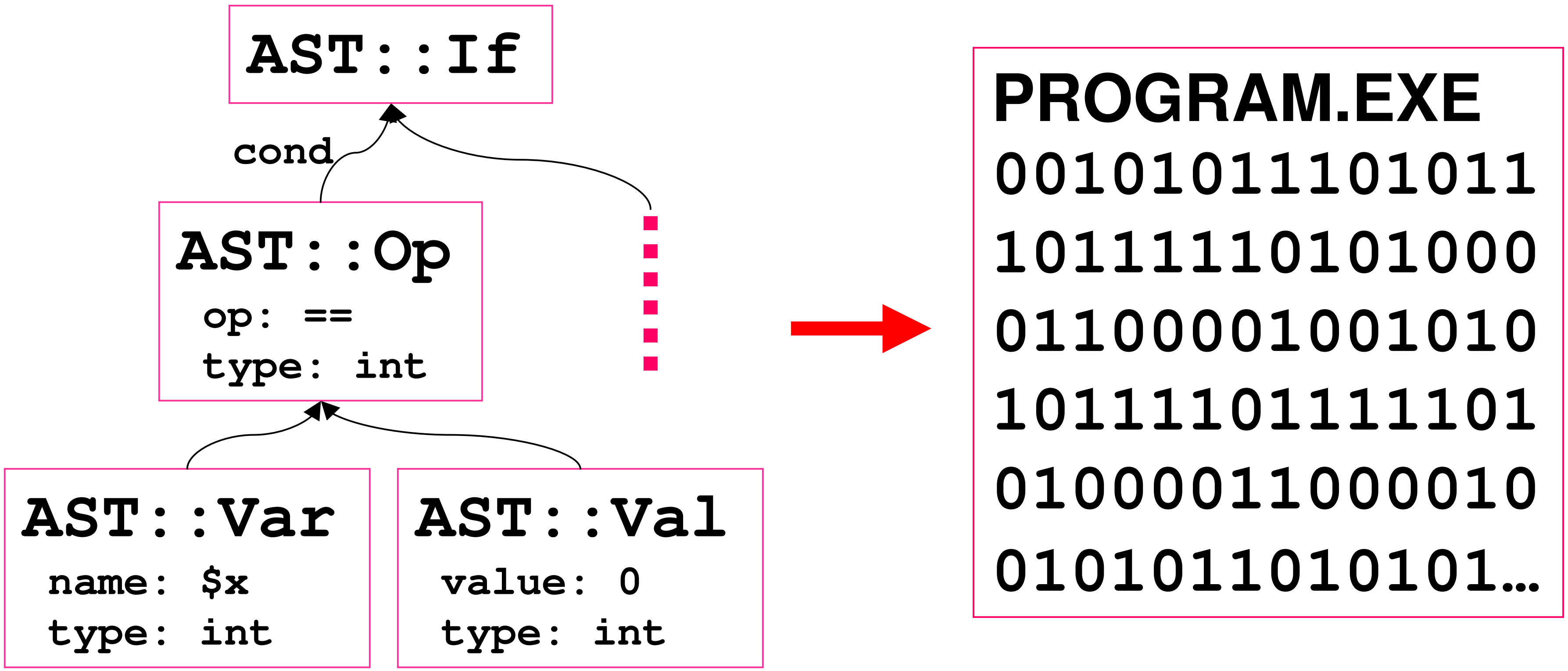
4. If we're Perl 5, we'll now walk over that tree and, for each node, do something




Alternate
Reality

The Journey Of A Program

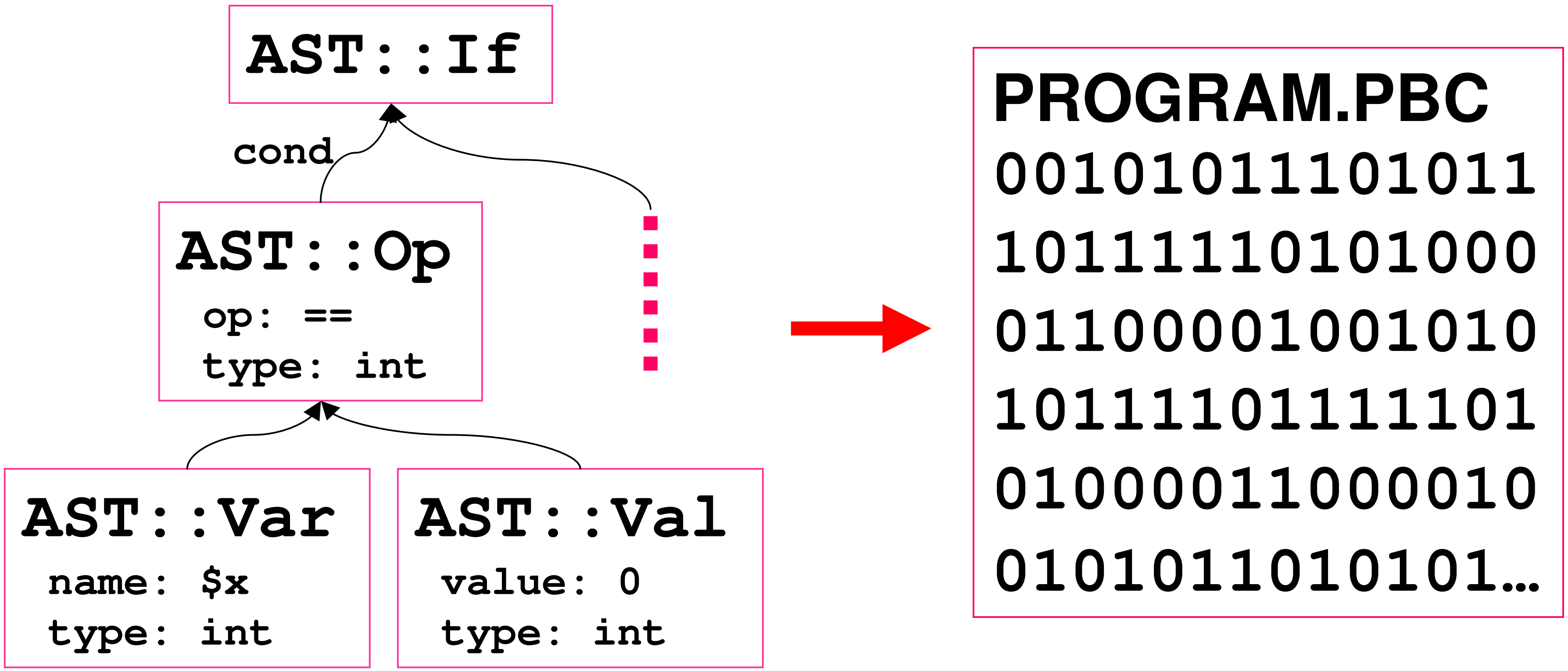
4. We walk over the tree and generate machine code for each node




Alternate
Reality

The Journey Of A Program

4. We walk over the tree and generate bytecode for a virtual machine



The Journey Of A Program

5. A virtual machine (such as the JVM or Parrot) interprets the bytecode or JIT-compiles it to machine code

PROGRAM.PBC

```
00101011101011
10111110101000
01100001001010
10111101111101
01000011000010
0101011010101...
```

Grammars

A Detour Into Linguistics

- Linguists have been analysing real languages for longer that we've had programming languages to consider
- One of the many things they came up with was the idea of a grammar
- Essentially, defining a language as a set of rules; too rigid and formal to really work for natural language, but great for programming languages!

Grammars

- Grammars are concerned with syntax, not meaning
- The grammar for a programming language can be used to generate all syntactically valid programs for that language
- **A grammar is a formal way of defining the syntax for a language**

A grammar is made up of…

- Terminals – things that we see in the language itself

```
digit ::= \d+
op ::= + | - | * | /
```

- Production rules defining non-terminals

```
expr ::= digit op expr
       | digit
```

- Note rules can be recursive (beware of what recursion is allowed – it differs)

Generation With A Grammar

- We also define a start rule: in this case, we will use **`expr`**.

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

- A whole program is represented by this start rule.

Parsing

- Grammars are most commonly used to parse programs rather than generate them.
  - Take a program
  - Work out what grammar rules you need to get back to the start rule from the tokens the program is made up of

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

```
35 + 7
```

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

```
35 + 7
```

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

35 + 7

digit: 35

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

35 + 7

digit: 35

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

35 + 7

digit: 35

op: +

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

```
35 + 7
```

```
digit: 35    op: +
```

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

35 + 7

digit: 35	op: +	digit: 7

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

```
35 + 7
```

digit: 35	op: +	digit: 7

Parsing

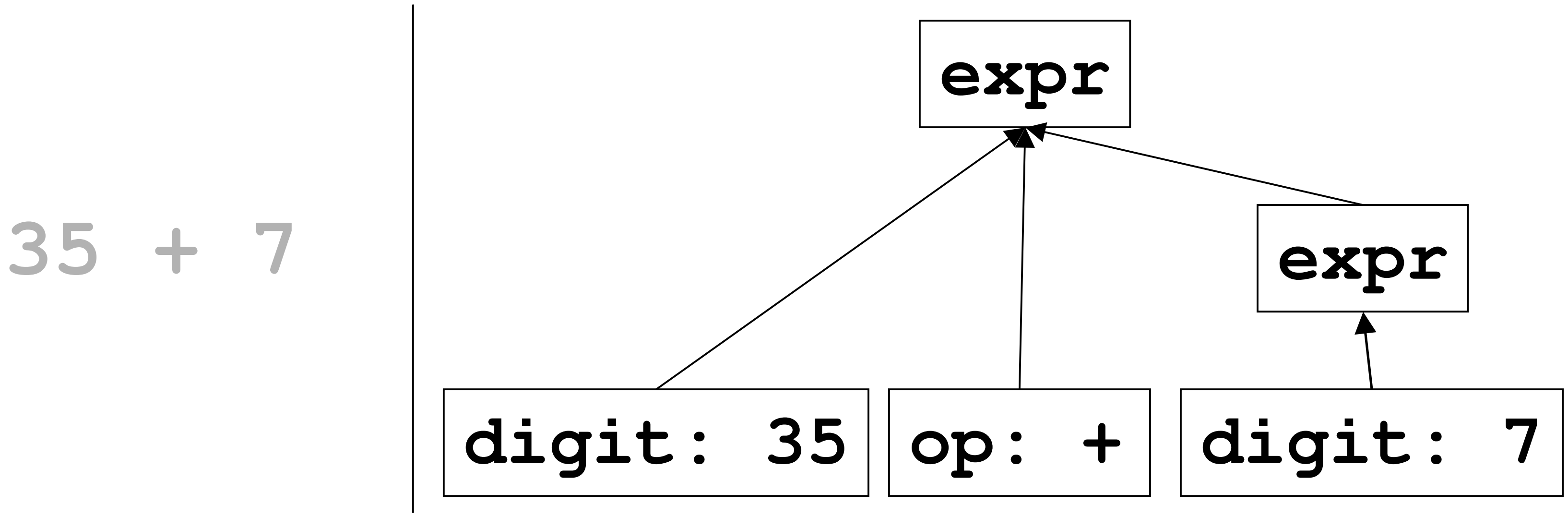
- Result is that we build a parse tree

```
expr ::= digit op expr
         | digit
digit ::= \d+
op ::= + | - | * | /
```

Parsing

- Result is that we build a parse tree

```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```



Grammars In Perl 6

- Can translate our example directly into Perl 6.

```
grammar Math {
    token op    { <'/'> | <'*'>
                | <'+'> | <'-'> }
    token digit { \d+ }
    token expr  { <digit> <op> <expr>
                | <digit> }
}

my $tree = "35+7" ~~ /^<Math.expr>$/;
```

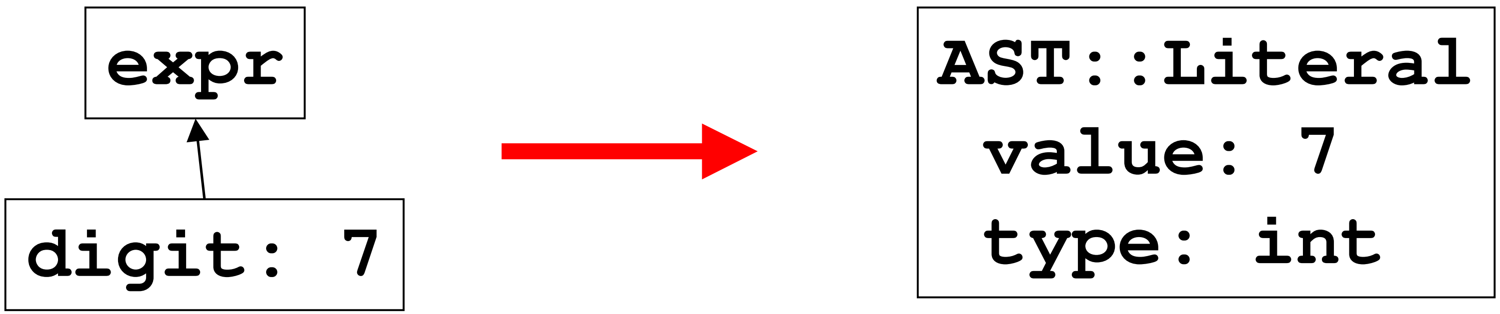
Attribute Grammars

Mostly A Scary Name

- Attribute grammars might sound less scary if we called them Tree Grammars
- They are used in the Tree Grammar Engine, part of the Parrot compiler tools
- Instead of taking a string of characters as input, tree grammars take a tree
- Specify a "transform" to perform on each type of node in the tree

Abstract Syntax Trees

- Aim is to capture the semantics, but without the mess in the parse tree that was a result of the language's syntax
- Also annotate nodes with extra stuff – perhaps types



Writing Attribute Grammar Transforms

- This is TGE-like syntax (you can't write Perl 6 to implement the transform yet, only PIR)
- Here's the rule for **`digit`** nodes

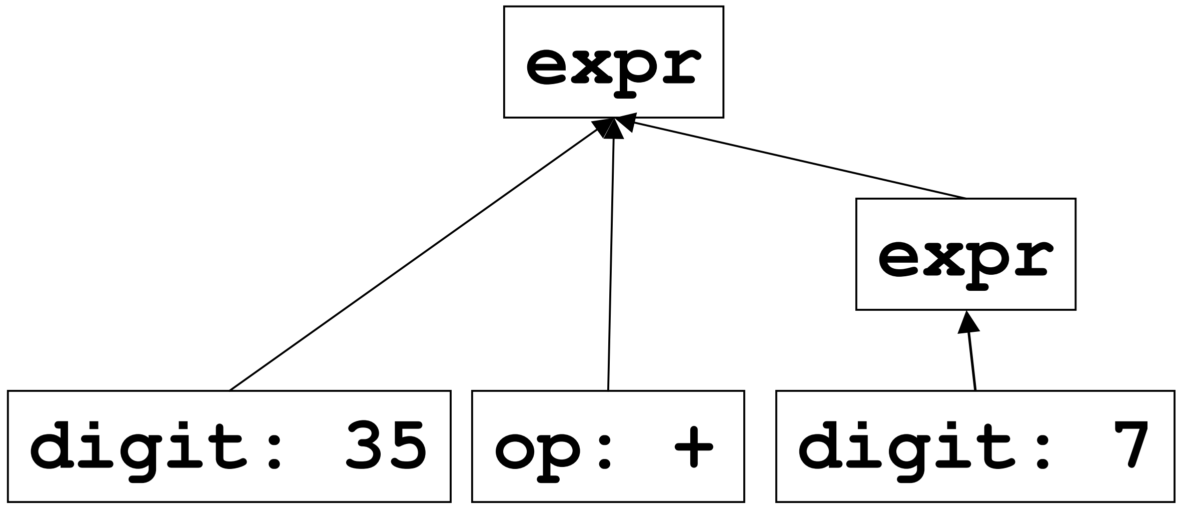
```
transform make_ast (digit) {
    $result = new AST::Literal;
    $result.value = $node;
    $result.type = 'int'
}
```

Writing Attribute Grammar Transforms

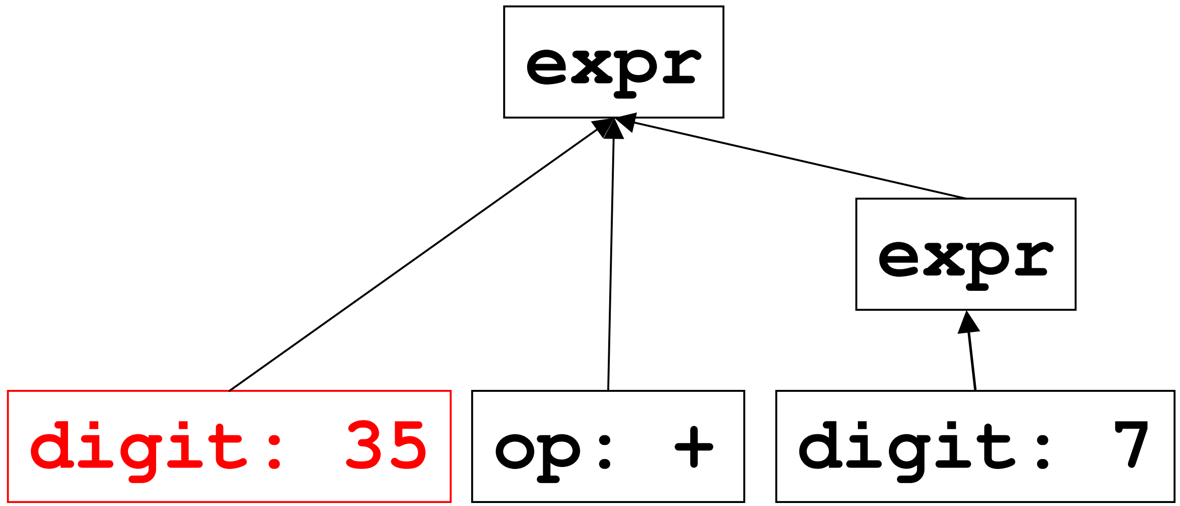
- The rule for **`expr`** is more complex

```
transform make_ast (expr) {
    if $node<op> {
        $result = new AST::Op;
        $result.opname = $node<op>;
        $result.oper1 = $node<digit>;
        $result.oper2 = $node<expr>;
    } else {
        $result = $node<digit>;
    }
}
```

From Parse Tree To AST

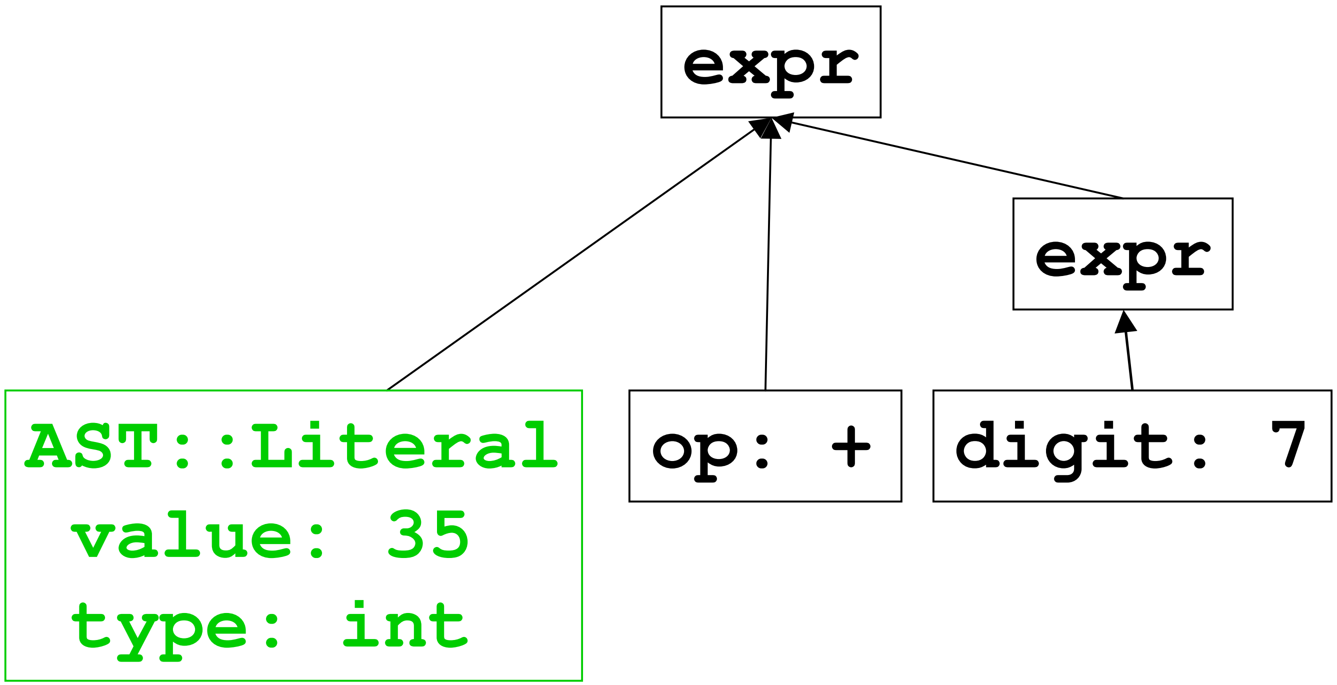


From Parse Tree To AST

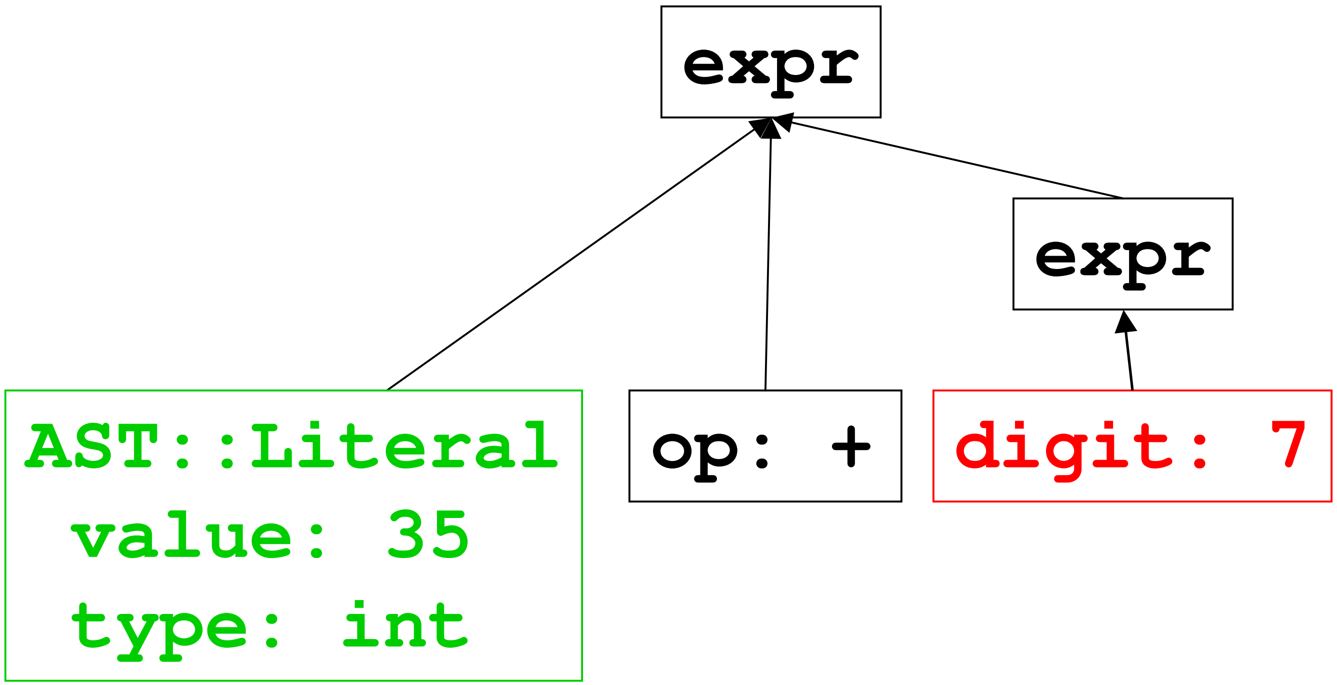



```
transform make_ast (digit)
```

From Parse Tree To AST

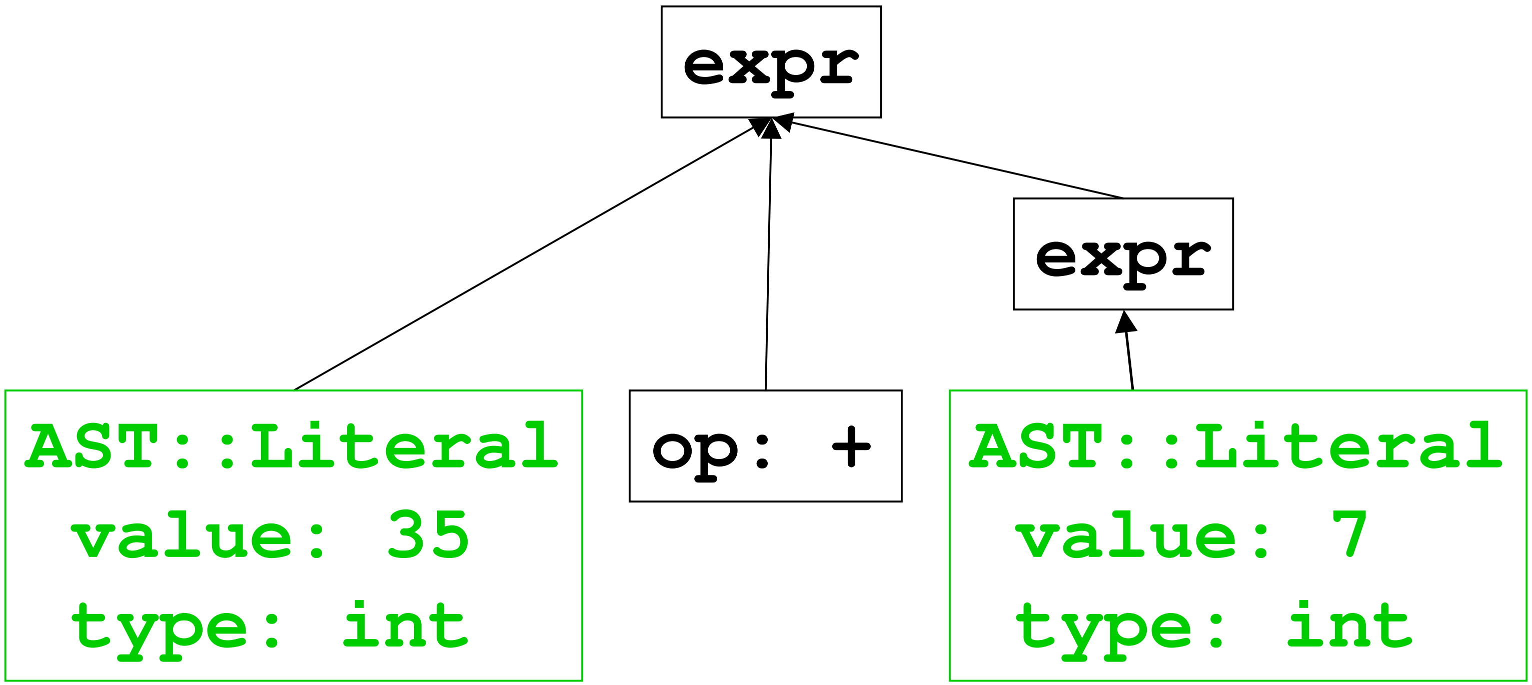


From Parse Tree To AST

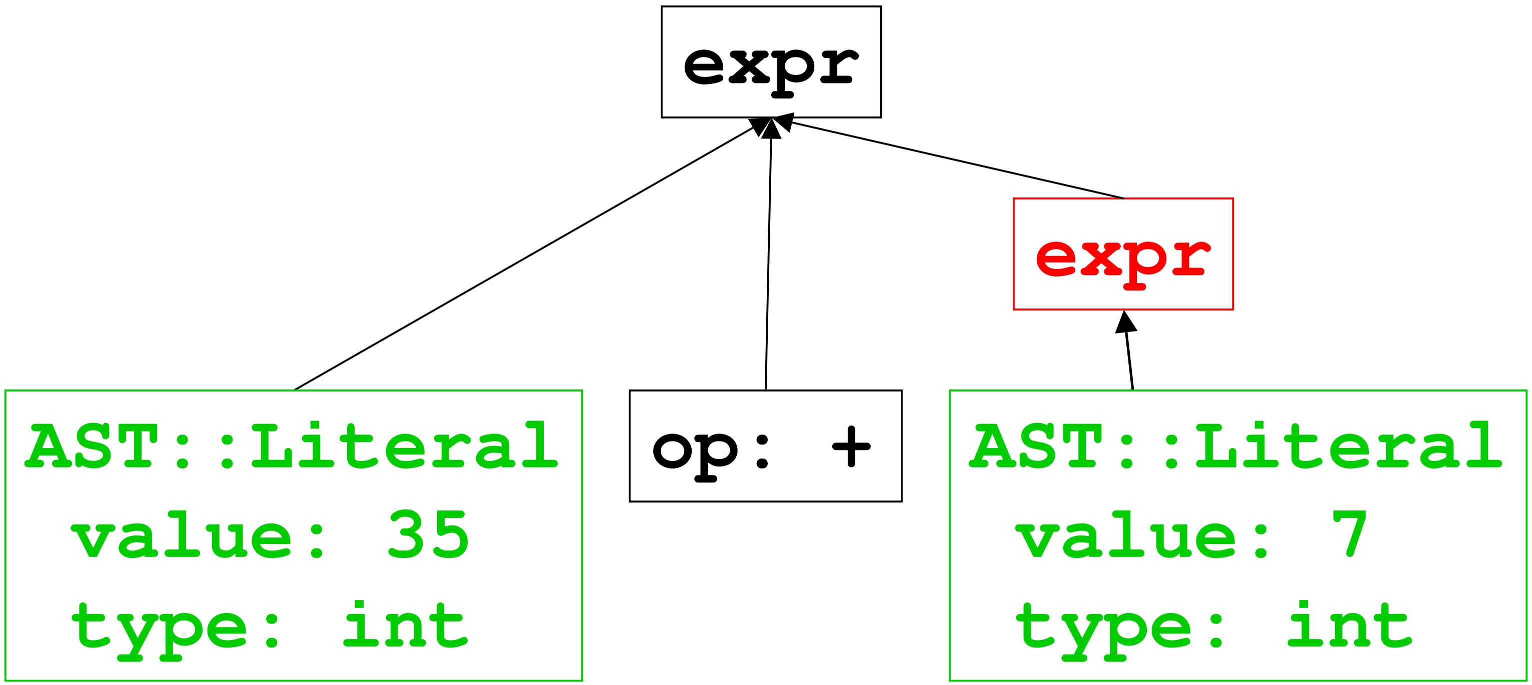


transform make_ast (digit)

From Parse Tree To AST

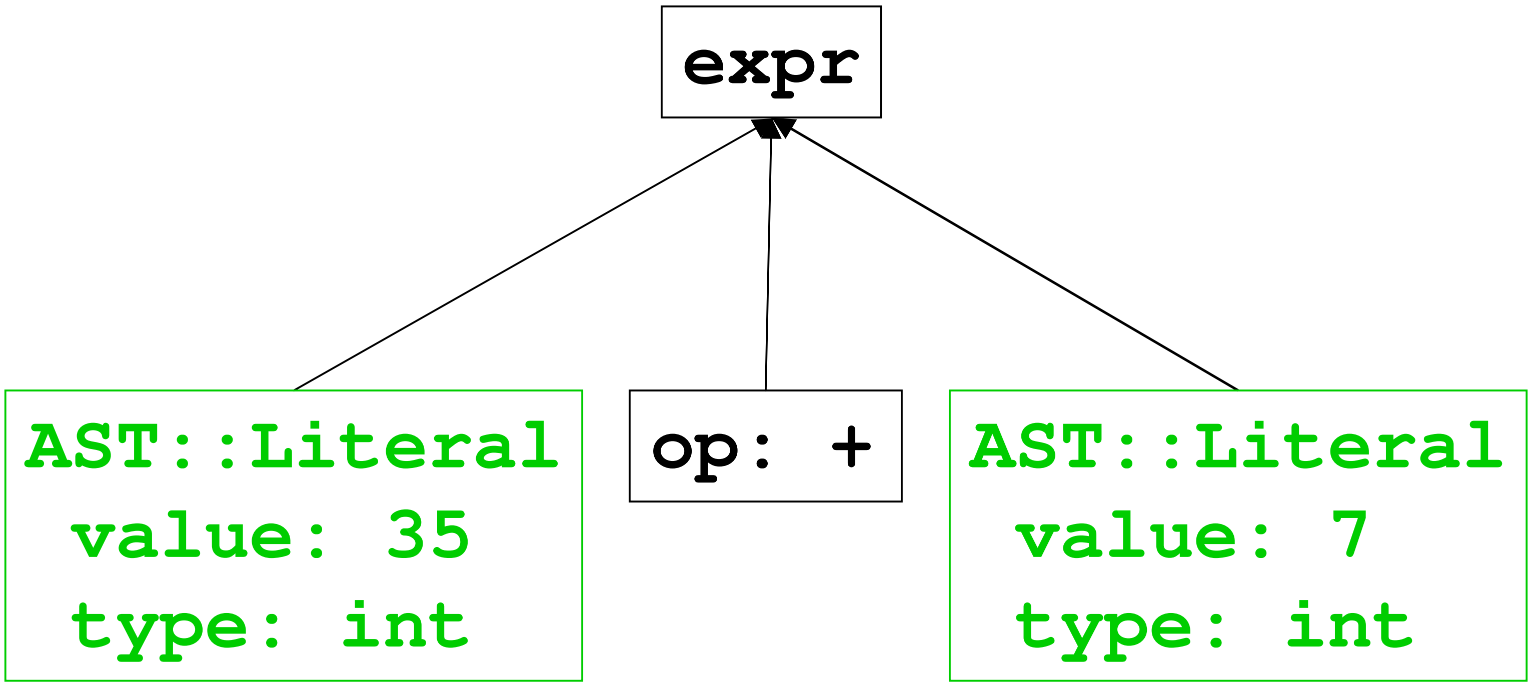


From Parse Tree To AST

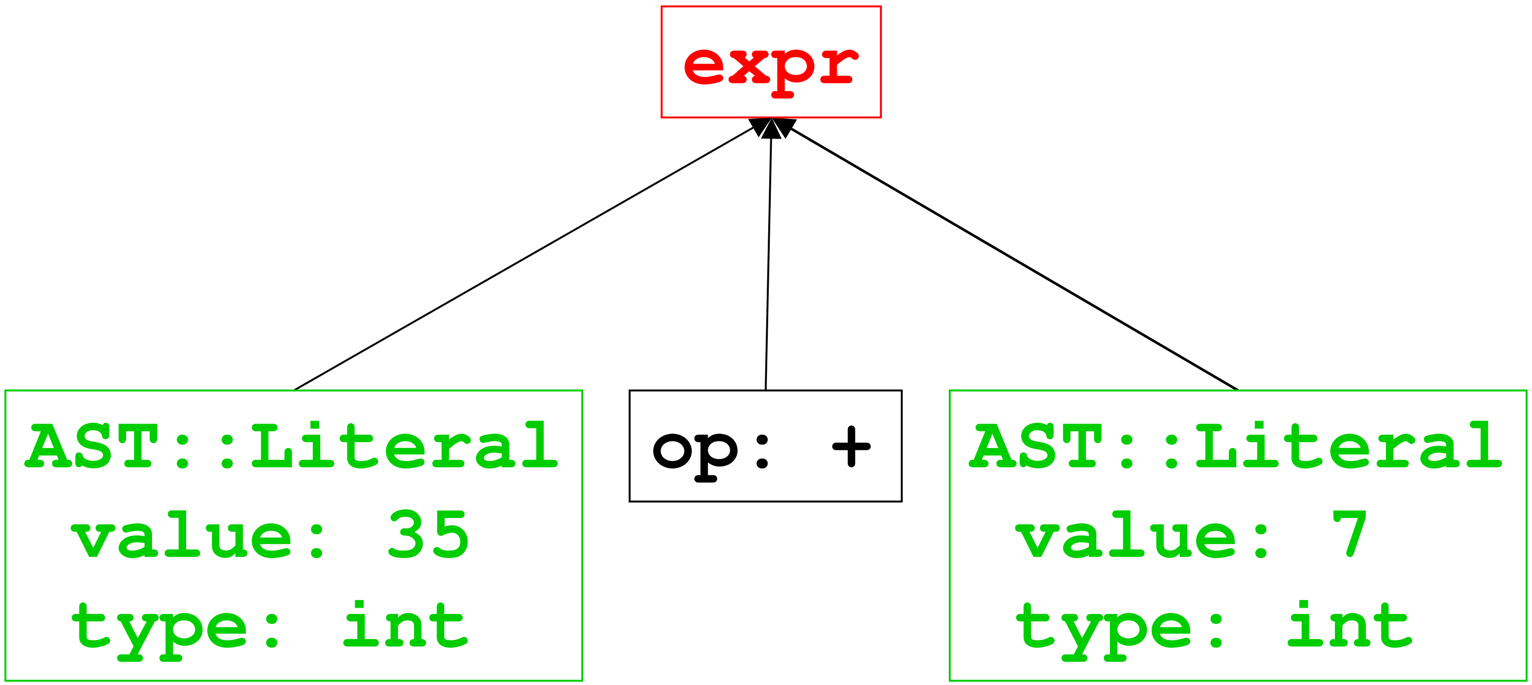



```
transform make_ast (expr)
```

From Parse Tree To AST

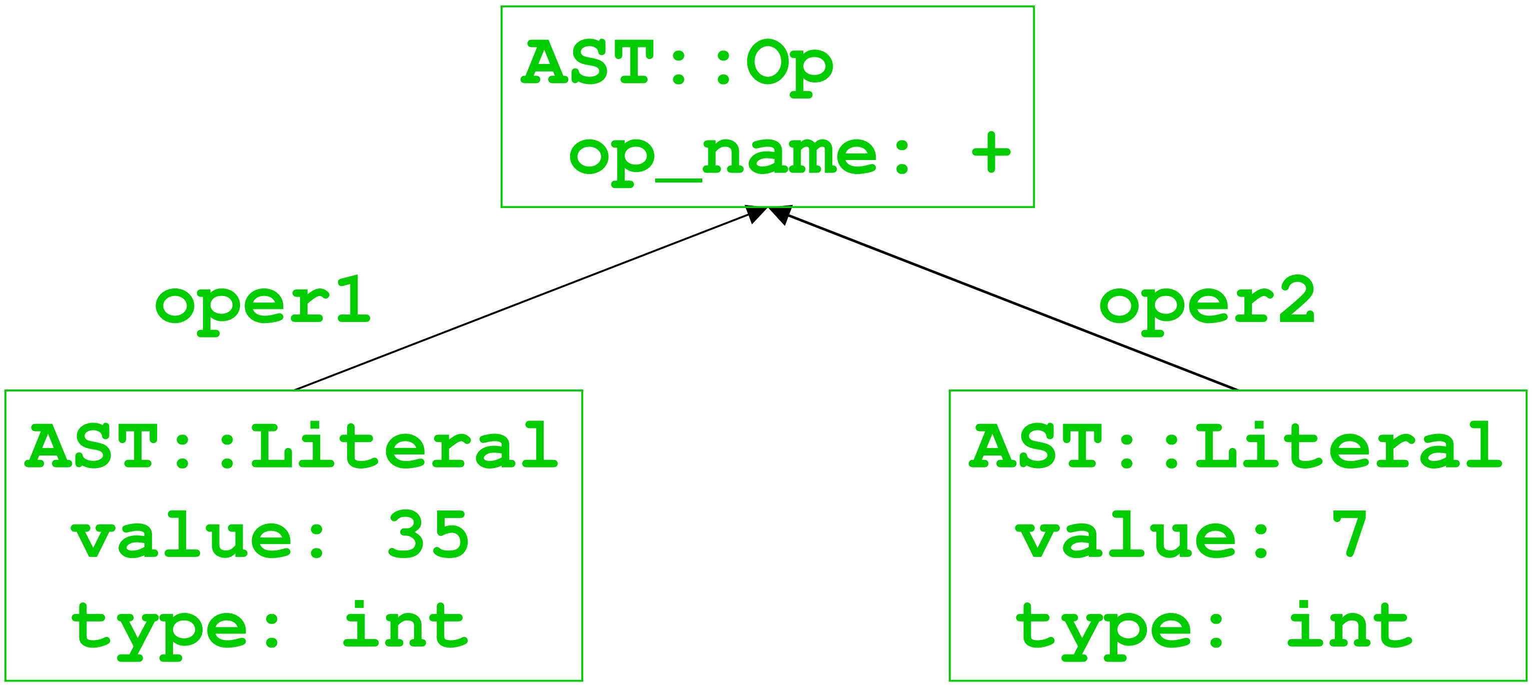


From Parse Tree To AST



`transform make_ast (expr)`

From Parse Tree To AST



Formal Semantics

Oh, behave!

- Grammars enabled us to formally specify the syntax of a language
- Formal semantics is about formally specifying the behaviour of the language

Operational Semantics

- We formalize the execution of the program by taking steps according to a sequence of evaluation rules
- These evaluation rules are what formally define the language
- In the examples I will demonstrate, at any point in the execution we will have the current term that is being evaluated and a store (mapping names to values)

Operational Semantics

- We will take a very simple language to define the semantics for
- It's helpful to see the syntax first – here it is specified as a grammar

```
value ::= n | x | true | false
    (n is an integer, x is a name)
term ::= if expr then expr else expr
       | expr + expr
       | expr == expr
expr ::= value | term
```

Inductive Evaluation Rules

- Terms in our program fall into two categories
  - Things we can evaluate right away (for example, 39 + 3) – rules for these are our **base cases**
  - Things we need to evaluate part of first (for example, (27 + 12) + 3) – rules for these are our **inductive steps**

Inductive Evaluation Rules

- The key idea behind induction: we can always break a program down until we get to base cases
- This provides us with a mechanism for proving a semantics have a property:
  - Prove it for the base cases
  - Prove that inductive steps retain the property

Evaluation Rules – Base Cases

$$\frac{}{(n_1 + n_2, s) \rightarrow (n, s)} \text{ (when } n = n_1 + n_2\text{)}$$

$$\frac{}{(n_1 == n_2, s) \rightarrow (true, s)} \text{ (when } n_1 = n_2\text{)}$$

$$\frac{}{(n_1 == n_2, s) \rightarrow (false, s)} \text{ (when } n_1 \neq n_2\text{)}$$

- s represents the store (mapping names to values)
- → represents a step of computation
- n, n_1 and n_2 represent integers

Evaluation Rules – Base Cases

$$\frac{}{(if\ true\ then\ t_1\ else\ t_2, s) \rightarrow (t_1, s)}$$

$$\frac{}{(if\ false\ then\ t_1\ else\ t_2, s) \rightarrow (t_2, s)}$$

- t_1 and t_2 represent other terms in the program
- Essentially, if the condition is true, the term as a whole reduces to the “then” cause, otherwise it reduces to the “else” clause

Evaluation Rules – Inductive Steps

$$\frac{(t_1, s) \rightarrow (t'_1, s)}{(t_1 + t_2, s) \rightarrow (t'_1 + t_2, s)}$$

$$\frac{(t_2, s) \rightarrow (t'_2, s)}{(n_1 + t_2, s) \rightarrow (n_1 + t'_2, s)}$$

- You can read the first rule as “if I have two terms added together, I do a step of evaluation on the first term”
- Note that these two rules encode that we evaluate left to right for addition!

Evaluation Rules – Inductive Steps

- The rest of the inductive steps pretty much follow this pattern
- Remember how in the grammar I carefully separated terms from values
- This means that our rules are deterministic – there is always at most one rule we can choose
- If no possible rule, the program is stuck

Example Evaluation

- Here is an example evaluation using the rules that we defined.

```
(if x == 0 then 42 else 12, {x→0})
```

Example Evaluation

- Here is an example evaluation using the rules that we defined.

```
  (if x == 0 then 42 else 12, {x→0})
→ (if 0 == 0 then 42 else 12, {x→0})
```

Example Evaluation

- Here is an example evaluation using the rules that we defined.

```
   (if x == 0 then 42 else 12, {x→0})
→ (if 0 == 0 then 42 else 12, {x→0})
→ (if true then 42 else 12, {x→0})
```

Example Evaluation

- Here is an example evaluation using the rules that we defined.

```
   (if x == 0 then 42 else 12, {x→0})
→ (if 0 == 0 then 42 else 12, {x→0})
→ (if true then 42 else 12, {x→0})
→ (42, {x→0})
```

An Evaluation That Gets Stuck

- Evaluating this will get to a state where no rules apply

```
(if x + 5 then 42 else 12, {x→3})
```

An Evaluation That Gets Stuck

- Evaluating this will get to a state where no rules apply

```
   (if x + 5 then 42 else 12, {x→3})
→ (if 3 + 5 then 42 else 12, {x→0})
```

An Evaluation That Gets Stuck

- Evaluating this will get to a state where no rules apply

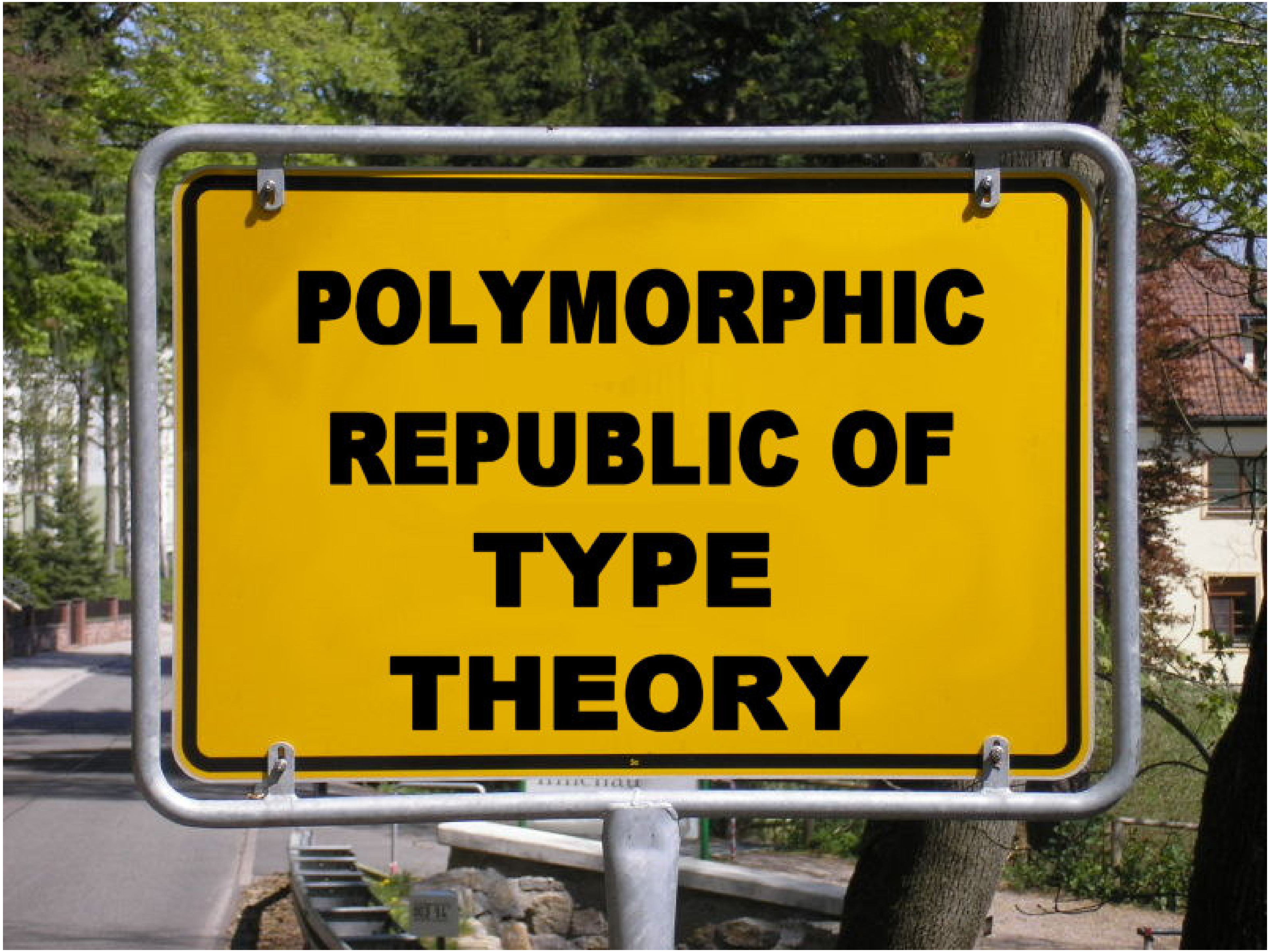
```
   (if x + 5 then 42 else 12, {x→3})
→ (if 3 + 5 then 42 else 12, {x→0})
→ (if 8 then 42 else 12, {x→0})
```

An Evaluation That Gets Stuck

- Evaluating this will get to a state where no rules apply

```
   (if x + 5 then 42 else 12, {x→3})
→ (if 3 + 5 then 42 else 12, {x→0})
→ (if 8 then 42 else 12, {x→0})
→ FAIL
```

- Would like to turn down programs like this somehow at compile time


POLYMORPHIC
REPUBLIC OF
TYPE
THEORY

What Is A Type?

- TMTOWTDI (There's More Than One Way To Define It)
- A common definition: a type classifies a value (e.g. 42 is an integer, "monkey" is a string…)
- Another definition: a type defines the representation of and set of operations that can be performed on a value

What Is A Type System?

- Real programs consist of terms that compute values
  - "`29 + 13`"
- A type system classifies a term in a program according to the type of values that it will compute
  - "`29 + 13`" will have type "`integer`"
- Vary greatly between languages

Formalizing Types

- We usually specify that a term has a type by placing a colon between the two

$$42 : int$$

$$1 + 5 : int$$

$$true : bool$$

Type Environments

- A type environment, often written Γ (uppercase Greek letter gamma), maps names (of variables in languages that have them) to types
- For example, the following type environment tells us the types of the scalars $x and $b

$$\Gamma = \{\ \$x \rightarrow\ int,\ \$b \rightarrow\ bool\ \}$$

Type Environments

- The type environment Γ on the last slide allows us to determine the following typing:

$$\boxed{2 * \$x : int}$$

- Formally we write this as follows:

$$\boxed{\Gamma \vdash 2 * \$x : int}$$

- Which we read as “gamma proves that 2 * $x has type int”

Inductive Typing Rules

- We use inductive rules, just like we did with operational semantics
- Here are some the base cases for our type system – the types for values

$$\overline{\Gamma \vdash n : int} \text{ (provided } n \text{ is an integer)}$$

$$\overline{\Gamma \vdash true : bool}$$

$$\overline{\Gamma \vdash false : bool}$$

$$\overline{\Gamma \vdash x : T} \text{ (provided } \Gamma(x) = T)$$

Inductive Typing Rules

- Addition could have this typing rule:

$$\frac{\Gamma \vdash t_1 : int \quad \Gamma \vdash t_2 : int}{\Gamma \vdash t_1 + t_2 : int}$$

- You can read this as “we can prove that $t_1 + t_2$ has type int provided that t_1 has type int and t_2 has type int”
- The conditions above the line must be true for the what is below the line to be

Inductive Typing Rules

- The typing rule for "if" is a little more complex; we introduce a type variable T:

$$\frac{\Gamma \vdash t_1 : bool \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash if\ t_1\ then\ t_2\ else\ t_3 : T}$$

- This specifies that the condition of the if statement must be a boolean and the branches of the if must have the same type (not true of all languages!)

Type Checking

- Given a type environment, a term and the type that we believe the term to have, type checking verifies that the term does indeed have that type

Given a type environment Γ, a term t and a type T, show that $\Gamma \vdash t : T$

- By doing type checking at compile time with the typing rule for “if” shown on the last slide, our stuck example from earlier is now rejected at compile time!

Polymorphism

- Again, TMTOWTDI (both for D = Define and D = Do)
- One definition: polymorphism occurs when a term or value can be classified as having more than one type
- Another definition: polymorphism allows the same code to operate on values of different types

Polymorphism

- Many ways to achieve polymorphism
  - Subclassing
  - Parametric polymorphism (aka generics and parameterised types)
  - Refinement types
- We can formalize all of these
- Will just look at how we formalize subclassing and refinement types

Subclassing (Inheritance)

- Perl 6 has some nicer syntax for defining a subclass than Perl 5:

```
class Melon is Fruit {
    ...
}
```

- We formalize subclassing by adding a sub-typing rule that looks something like this (we really need to define "isa")

$$\frac{\Gamma \vdash t : S \qquad S \; isa \; T}{\Gamma \vdash t : T}$$

Refinement Types

- A refinement type is obtained by adding constraints to an existing type
- For example, the type EvenInt is a refinement of the Int type that only contains even integers
- In Perl 6, EvenInt would be defined like this:

```
subset EvenInt of Int where { $^n % 2 == 0 }
```

Refinement Types

- Can use a more refined type in place of a less refined one (e.g. EvenInt in place of Int)
- We formalize this using the denotation of the type, which is the set of all values of that type.

$$\frac{\Gamma \vdash t : S \qquad [\![S]\!] \subseteq [\![T]\!]}{\Gamma \vdash t : T}$$

The End

Danke!

Any questions?