# Formal Theory, Informally

**Jonathan Worthington**

Birmingham.pm

"I need rat poison and beer to drink."

"I need [rat poison] and [beer to drink]."

"I need [rat poison and beer] to drink."

## Informal

- Ambiguity in natural languages is often a source of terrible puns
- It is also a source of confusion

# Formal Theory, Informally

## Formal

- Describe stuff using maths and logic, not English sentences
- Mathematical notation is just another language
- However, it is formally defined, unlike English
- Enables us to say exactly what we mean, without ambiguity

## Theory

- Theoretical work on computation appeared before the first electronic computers
- Provides us with tools to understand what we're doing
- Provides new ideas that we can use in the real world - even if we don't see the use for them right away (for example, RSA public key cryptography)

# Formal Theory, Informally

## Informally

- This isn't a maths lesson
- We'll look at some stuff that's come out of the theory world...
- ...see how it helps us formally define real world stuff...
- ...and see practical uses of it.

# Programming Languages

## Programming Languages

- There's lots of theory that I could talk about
- I'm going to focus on the theory that helps us to build and understand programming languages and the tools that support our usage of them
- First of all: how does a program go from source code to actually being executed?

# The Journey Of A Program

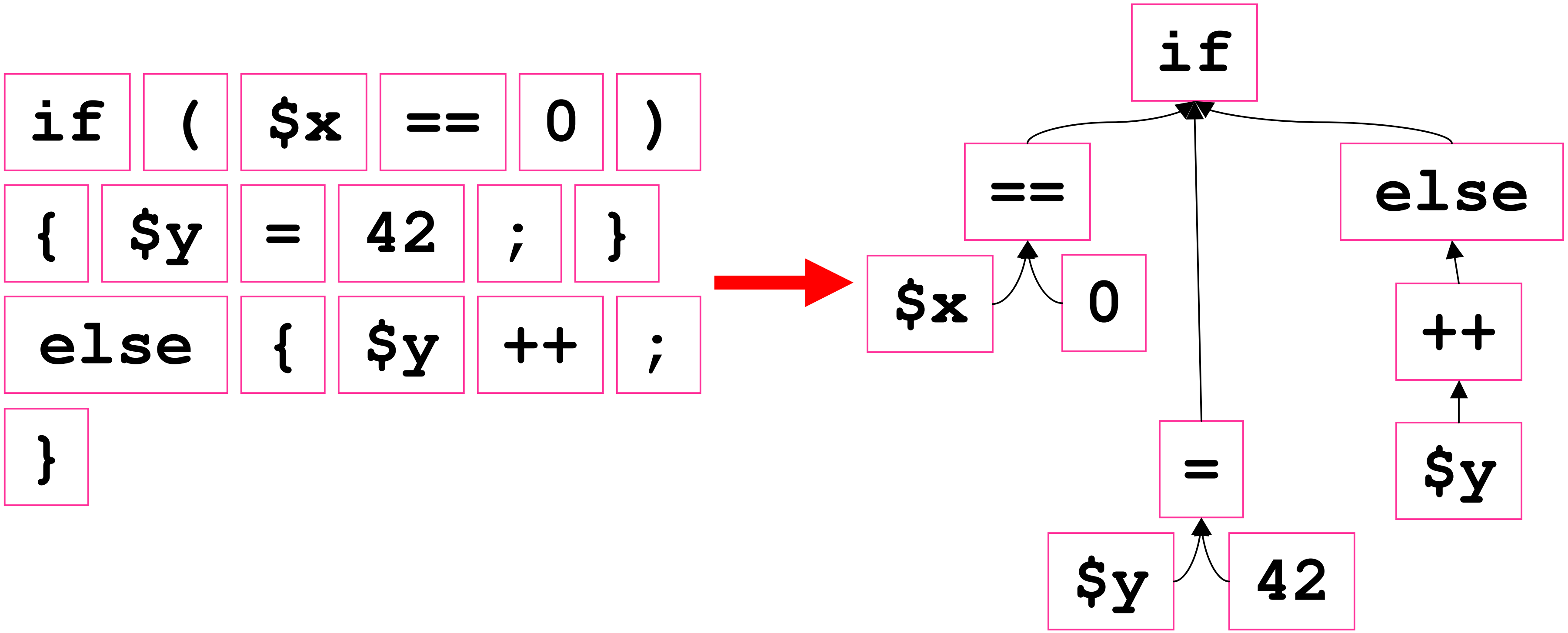
1. The program is tokenised

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

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

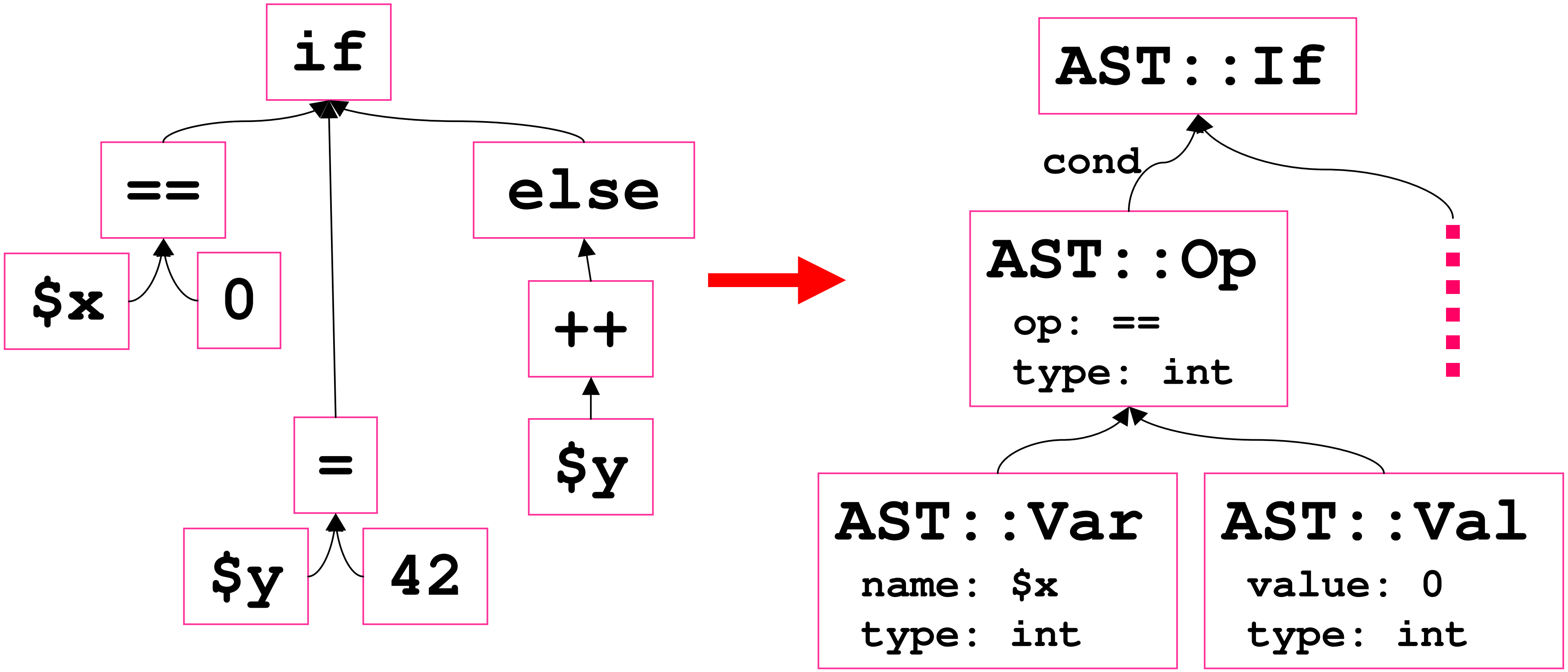
# The Journey Of A Program

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



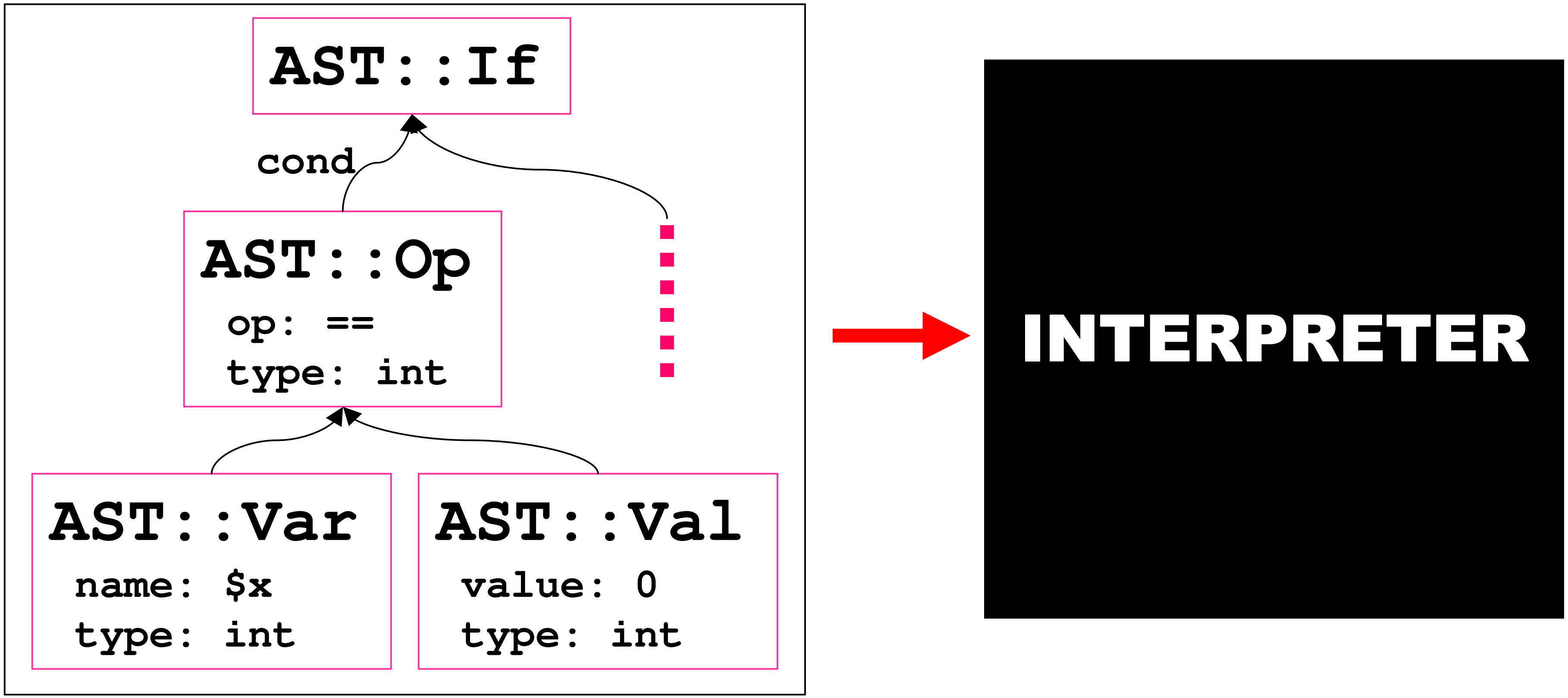
# The Journey Of A Program

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



# The Journey Of A Program

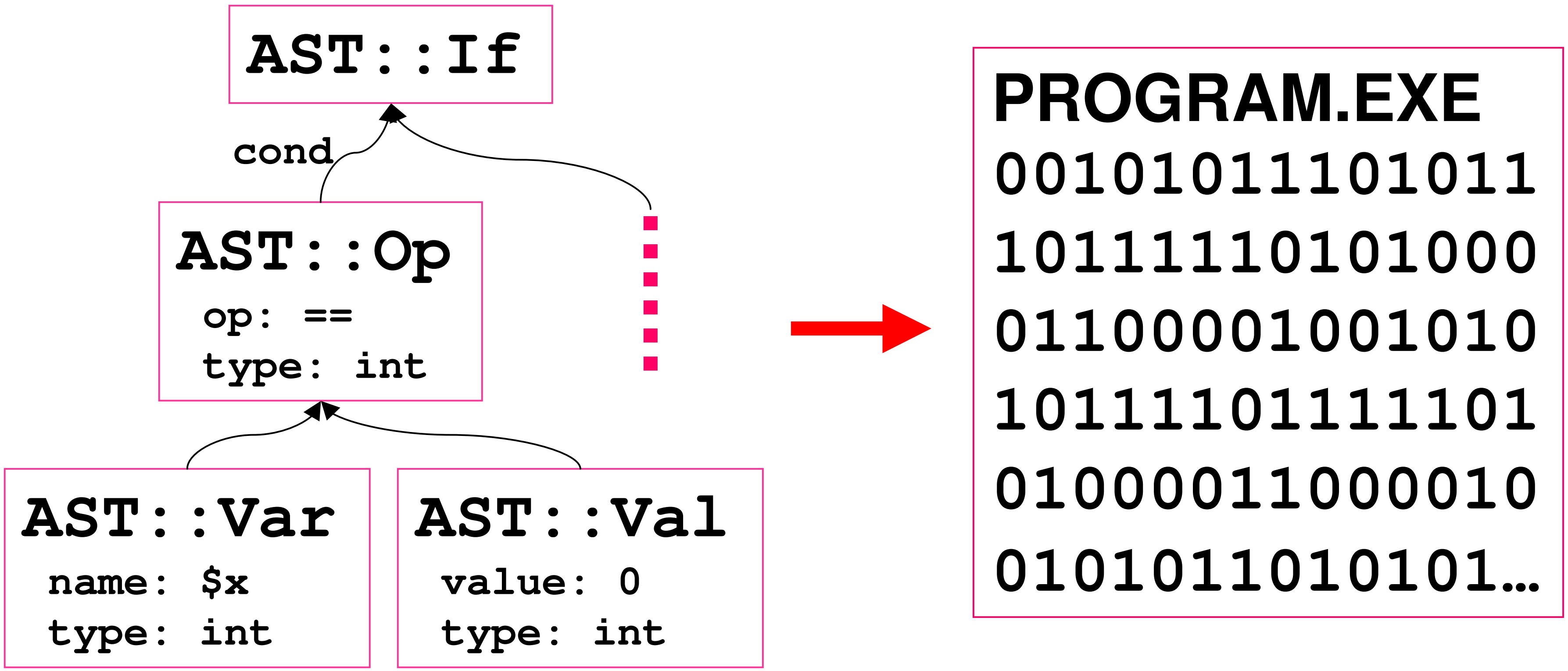
4. If we're Perl 5, we'll now walk over that tree and, for each node, do something




Alternate
Reality

# The Journey Of A Program

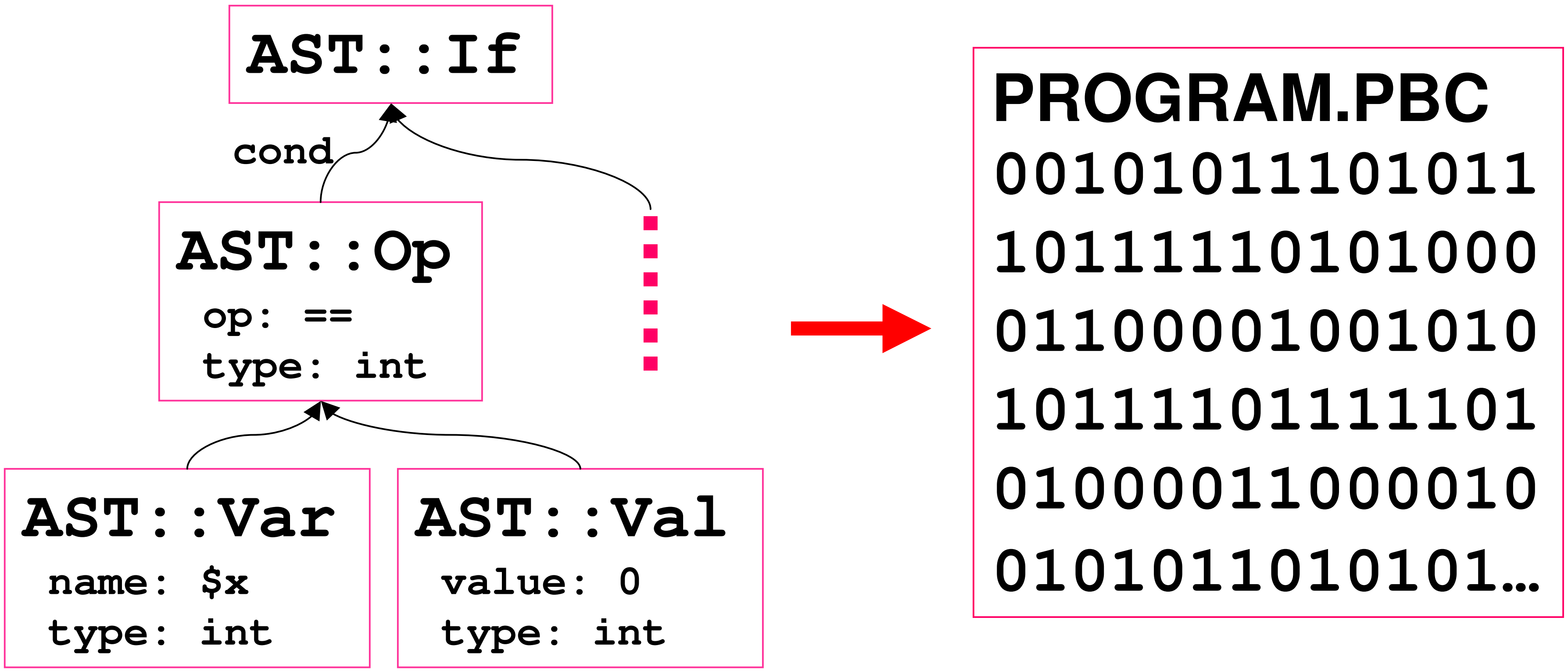
4. We walk over the tree and generate machine code for each node




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**

```
0010101110101 1
1011111010100 0
0110000100101 0
1011110111110 1
0100001100001 0
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**

## Grammars

- Just because a program is syntactically valid does not mean that it is meaningful

```
42 + "badger"
```

- Probably valid syntax as far as the grammar is concerned
- 42 in Perl, but still meaningless
- A compile-time type error in C#

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

```
digit op expr
```

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`digit op expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`digit op expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 op expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 op expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 op expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + expr`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + expr`**

# 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + digit`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + digit`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + digit`**

## 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 ::= + | - | * | /
```

- Can start expanding out the production rules until we reach all tokens.

**`41 + 1`**

## Parsing

- Grammars are more commonly used to do the reverse of this process
  - Taking a program
  - Work out what grammar rules you need to get back to the start rule
- There's more than one way to parse
  - Recursive descent
  - Stack/automata based

## 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

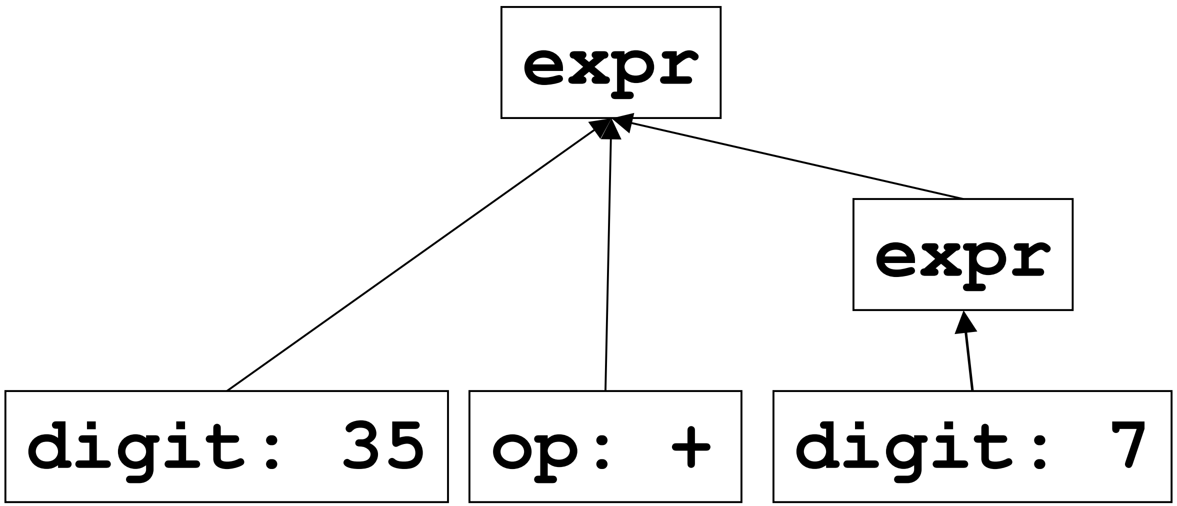
```
expr ::= digit op expr
       | digit
digit ::= \d+
op ::= + | - | * | /
```

## Parsing

- Result is that we build a parse tree

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

35 + 7



## Grammars In Perl 6

- So you're never going to write a compiler, and are wondering how grammars will be useful to you?
- Answer: Perl 6 has grammars built into the language!
- The syntax of the Perl 6 language itself is formally described by a grammar too, meaning that multiple implementations are now feasible

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

## Grammars In Perl 6

- Can translate our example directly into Perl 6.

```
# grammar Math { # Not yet in Pugs
    token op    { <'/'> | <'*'>
                | <'+'> | <'-'> }
    token digit { \d+ }
    token expr  { <digit> <op> <expr>
                | <digit> }
# }

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

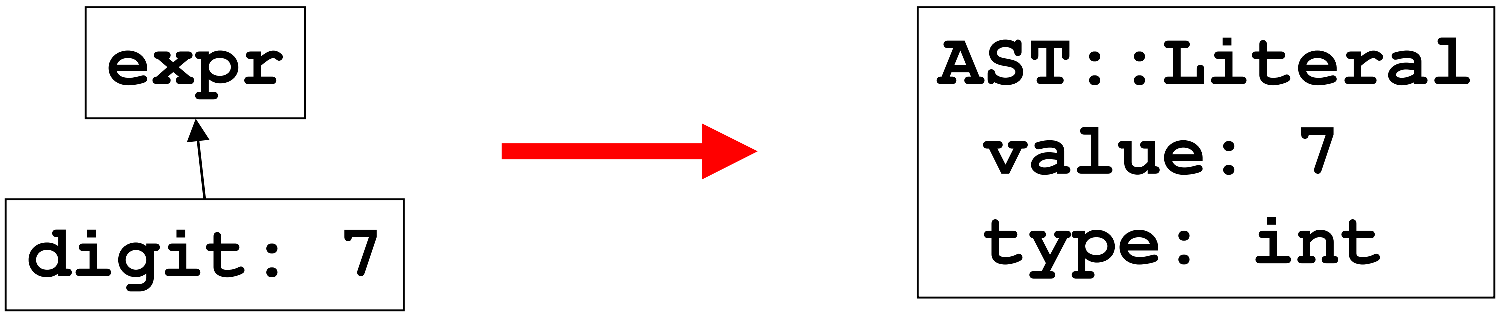
# Attribute Grammars

## 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

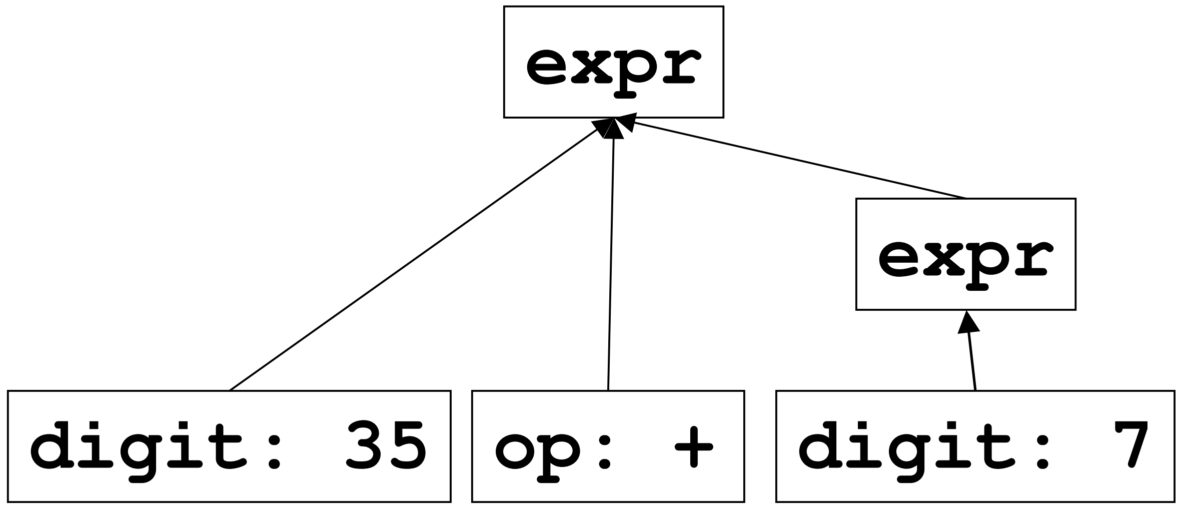
```
transform make_ast (digit) {
    my $result = new AST::Literal;
    $result.value = $node;
    $result.type = 'int'
}
```

## Writing Attribute Grammar Transforms

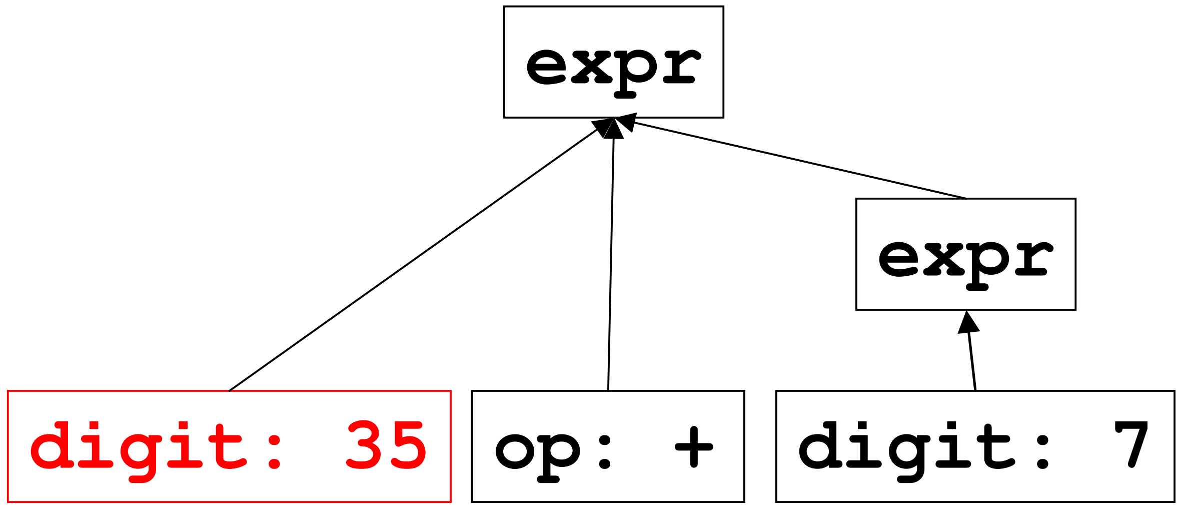
- The rule for **`expr`** is more complex

```
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

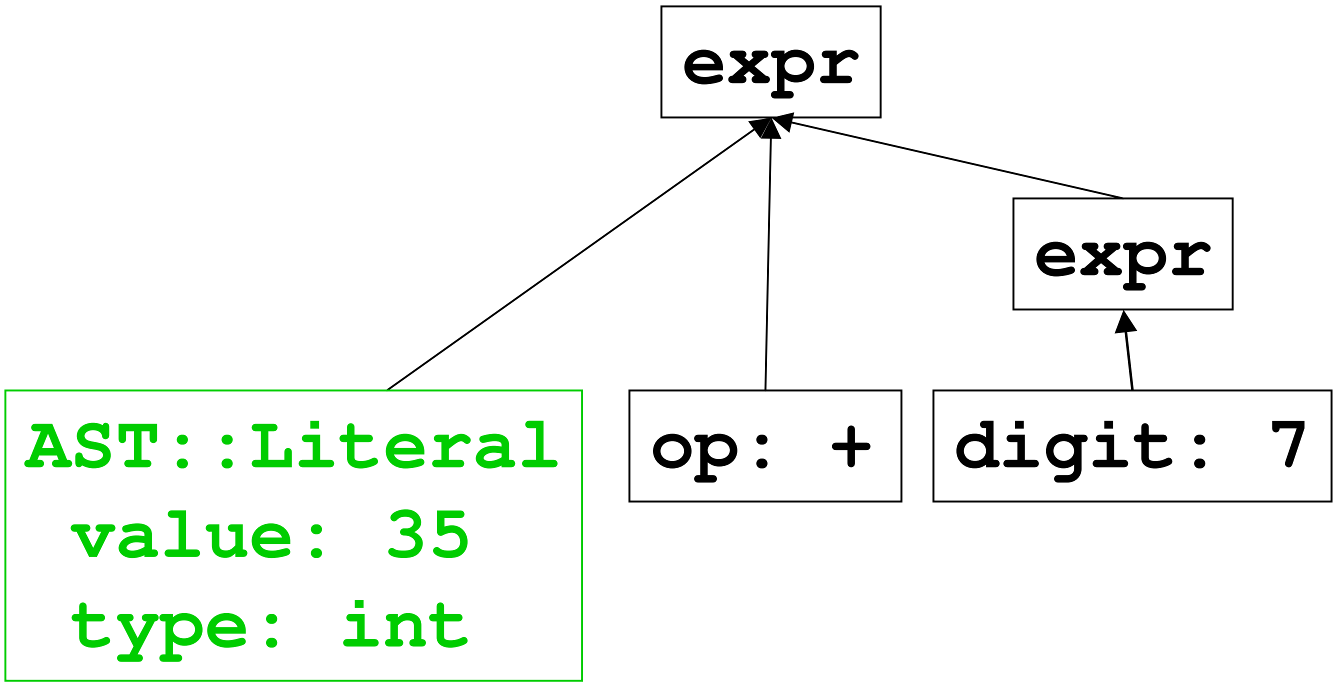


# From Parse Tree To AST

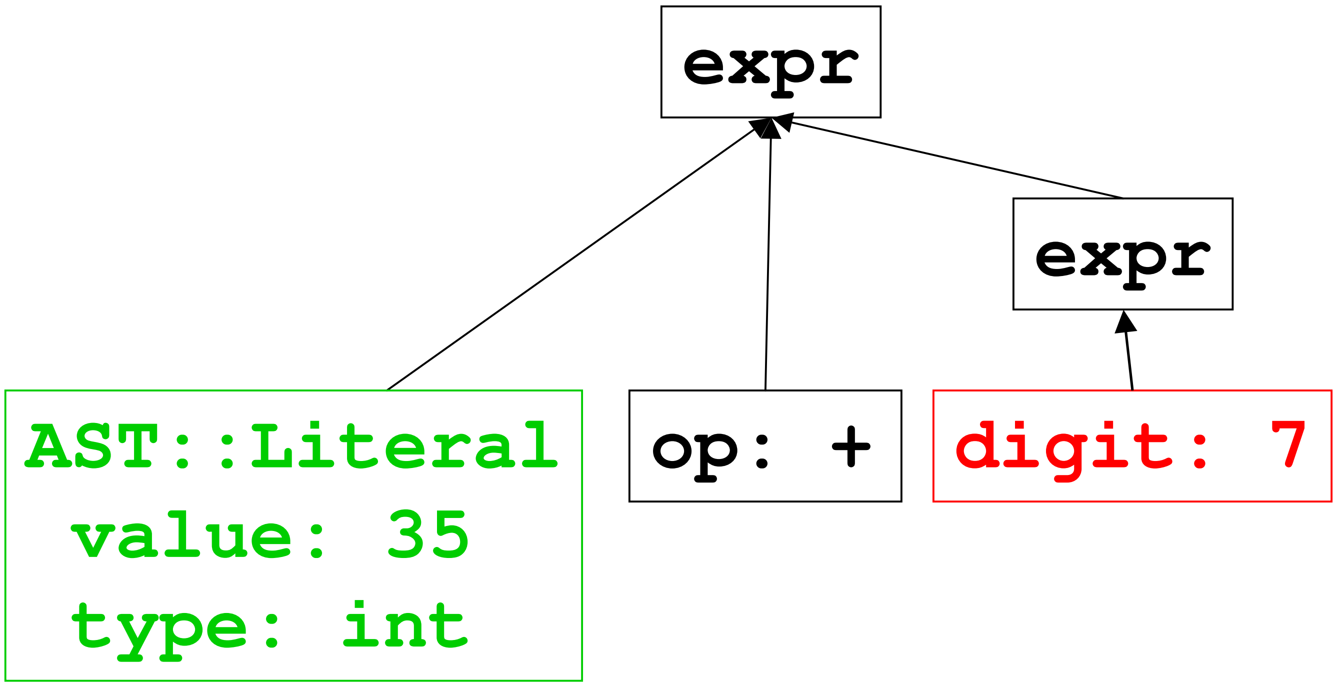



**`transform make_ast (digit)`**

# From Parse Tree To AST

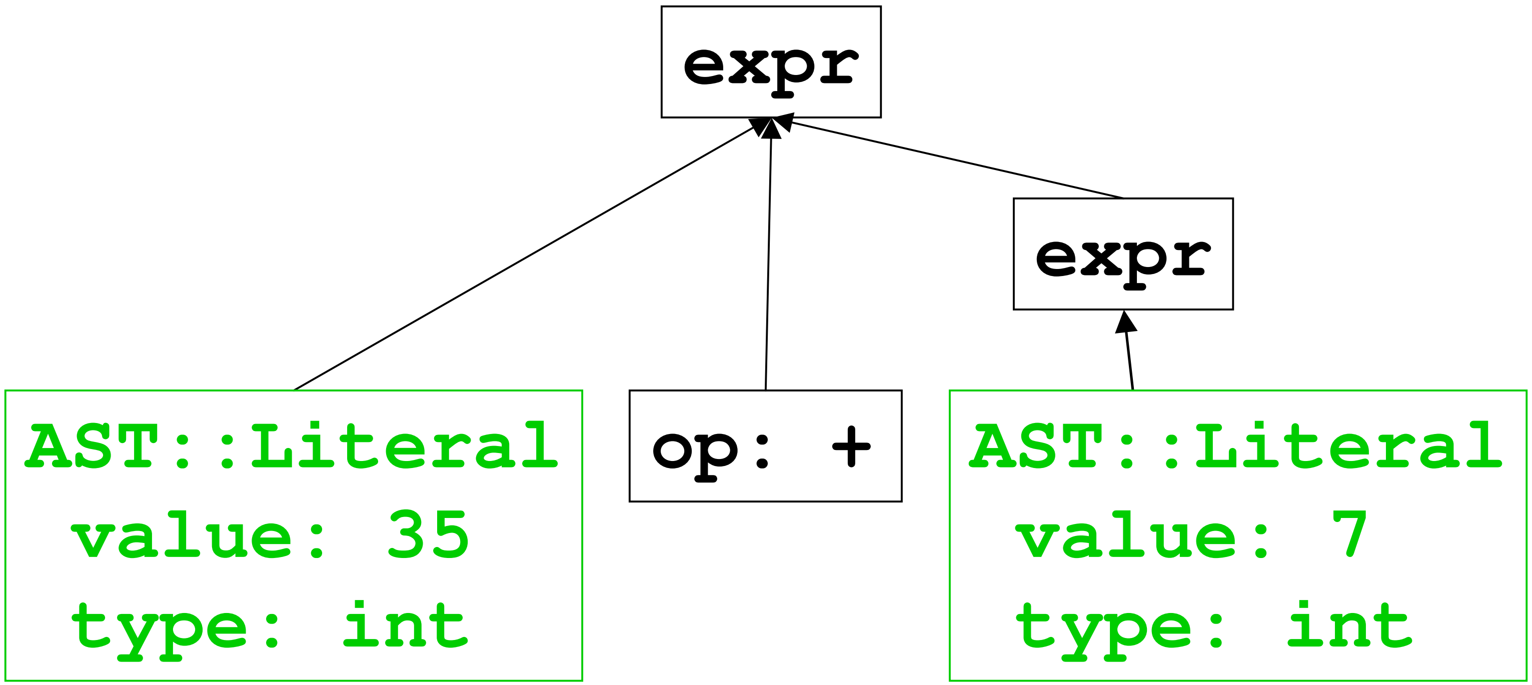


## From Parse Tree To AST

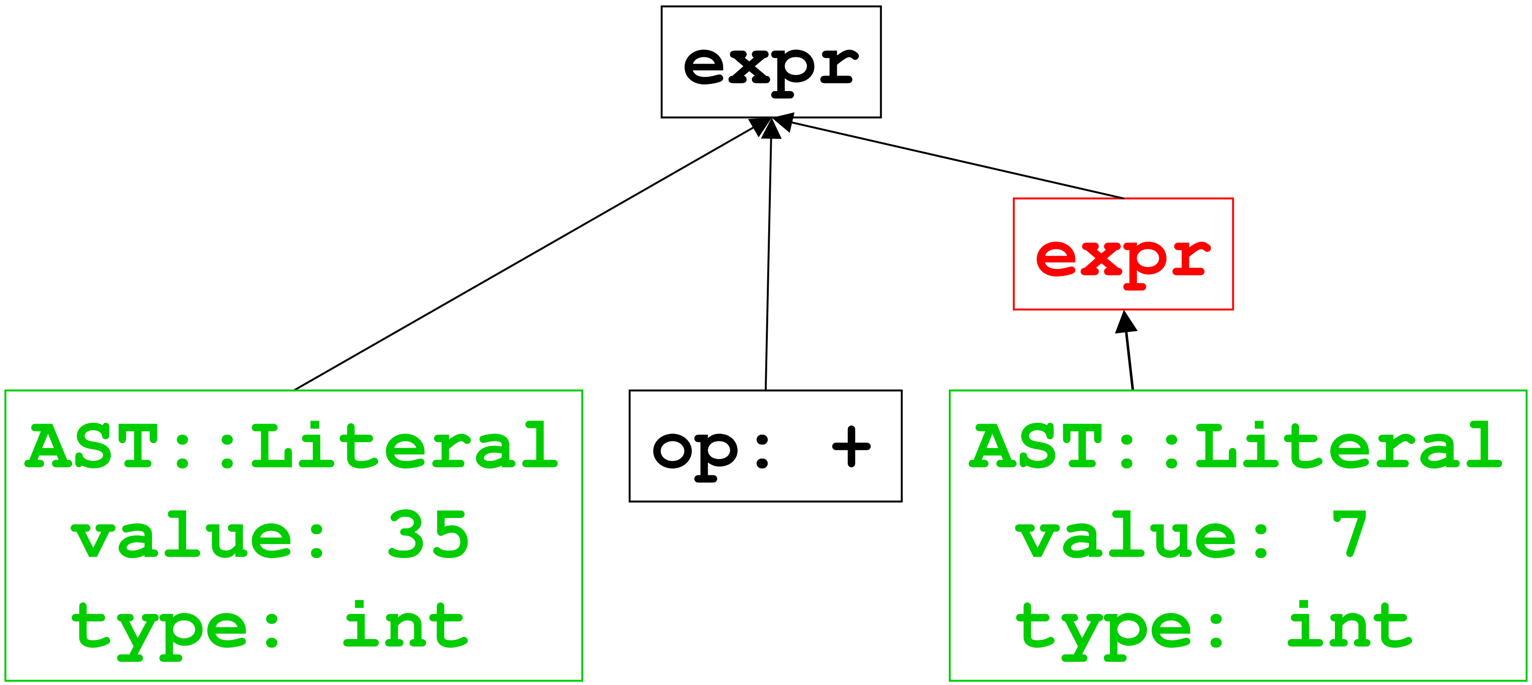



```
transform make_ast (digit)
```

# From Parse Tree To AST

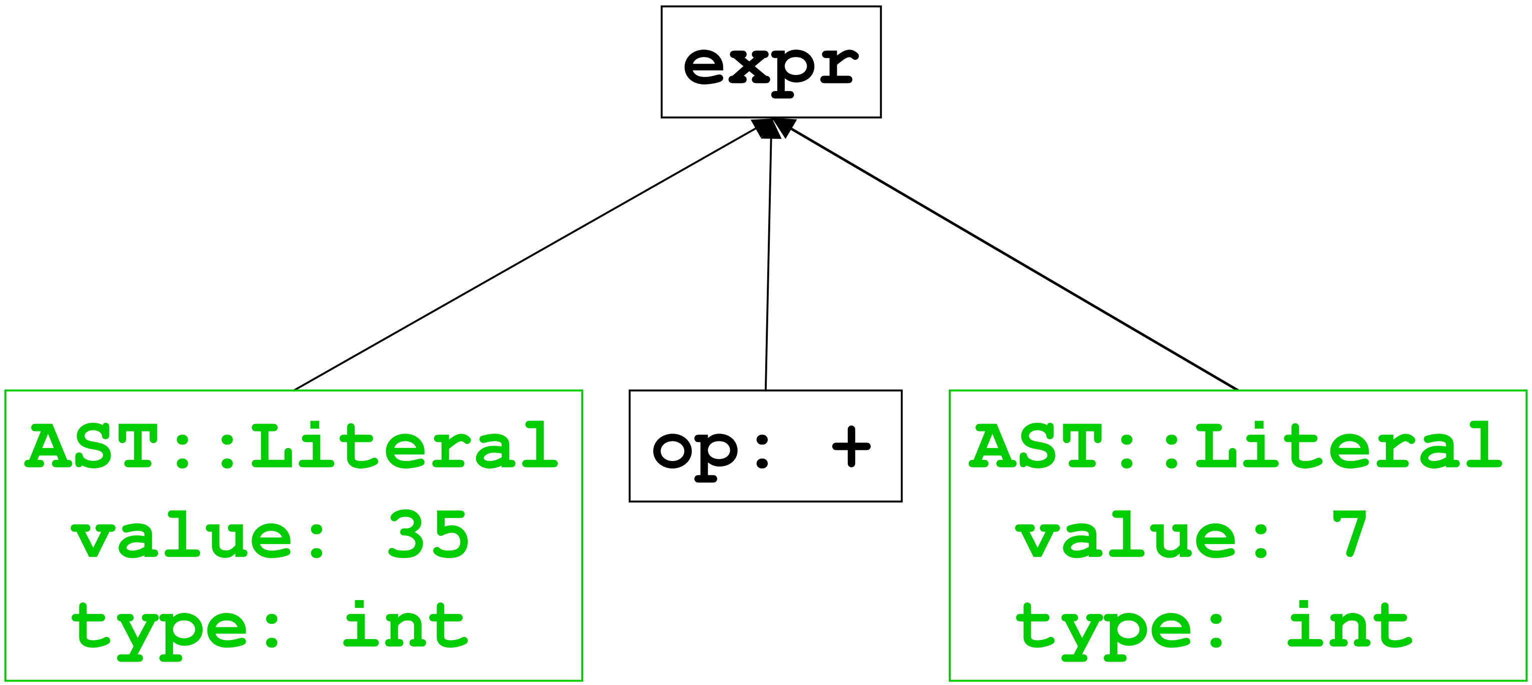


# From Parse Tree To AST

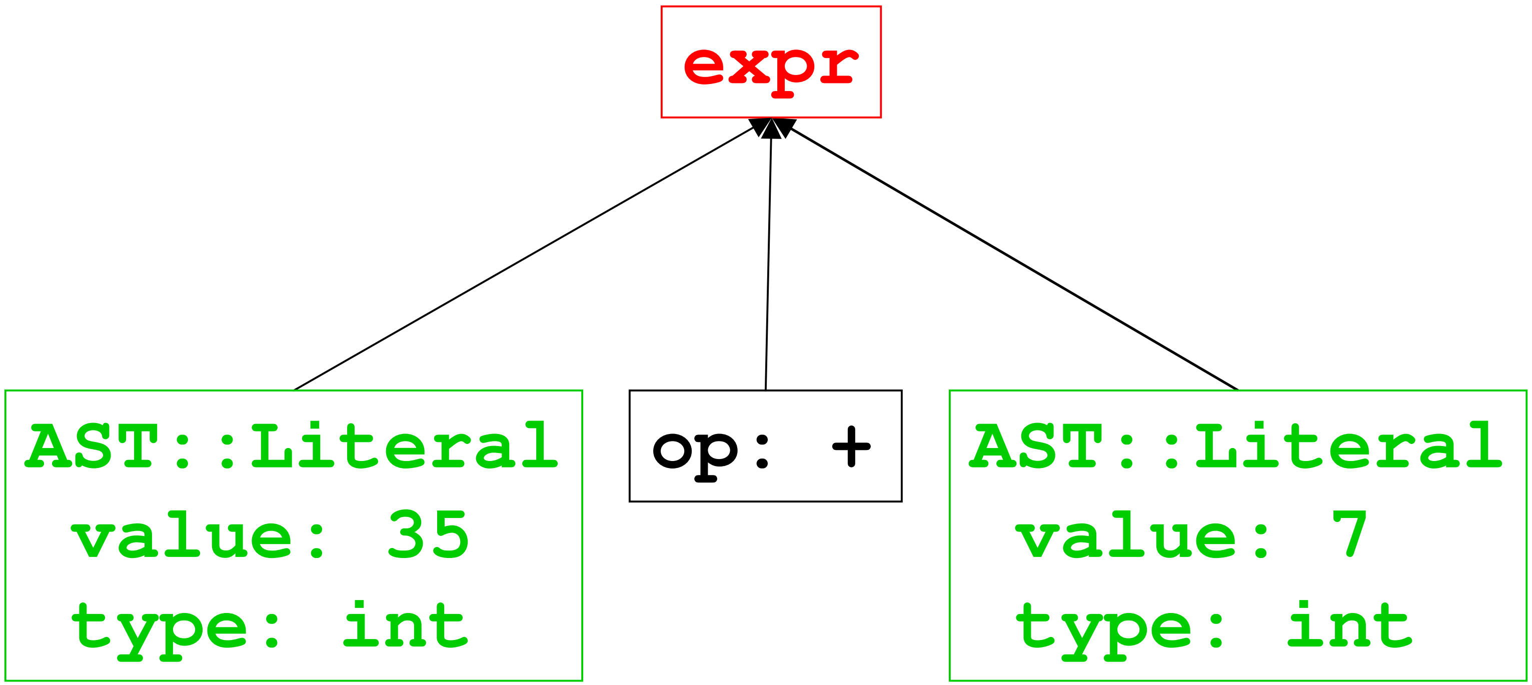



**transform make_ast (expr)**

# From Parse Tree To AST

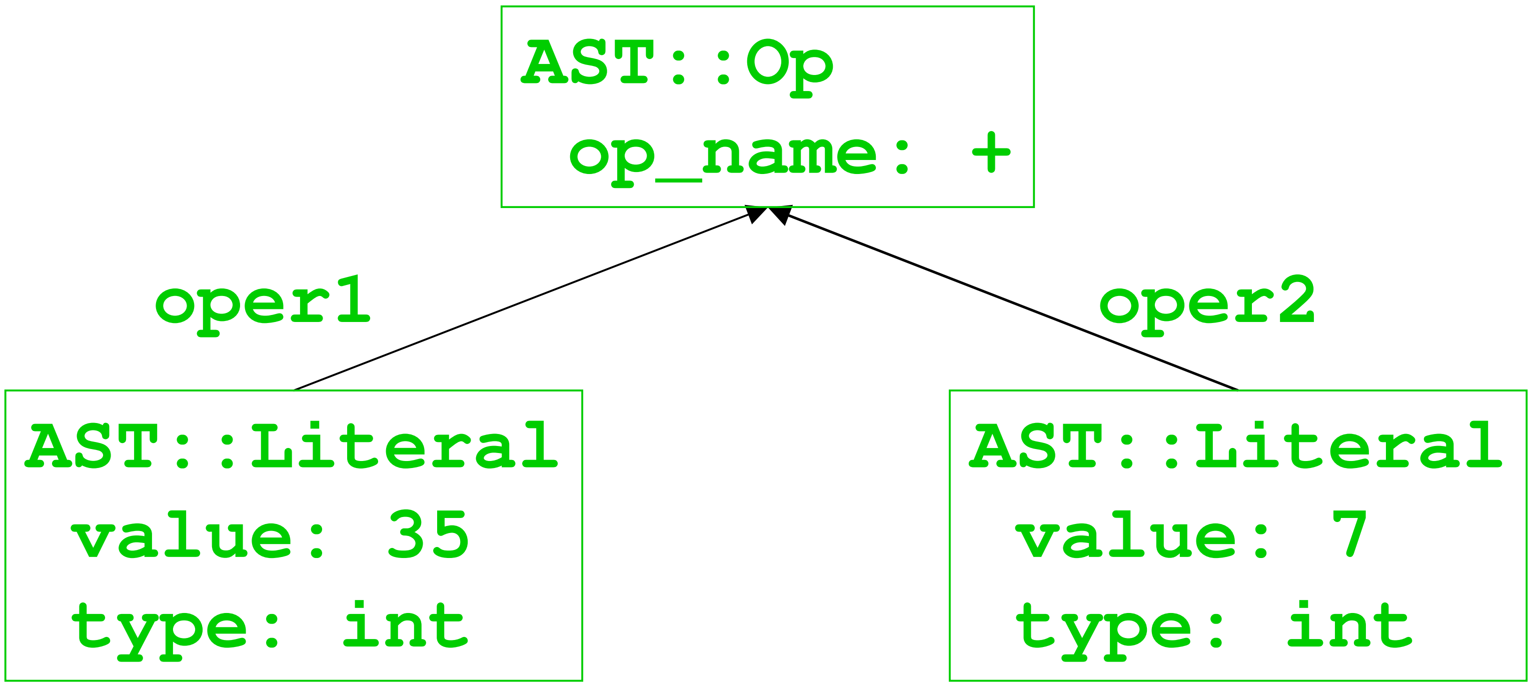


# From Parse Tree To AST




**`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

## Approaches To Formal Semantics

- **Operational semantics** describe the steps involved in executing the program. Syntax directed, quite easy to work with.
- **Denotational semantics** map the programming language onto a mathematical model. This is somewhat harder to work with.
- There are other approaches

## 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 -

```
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

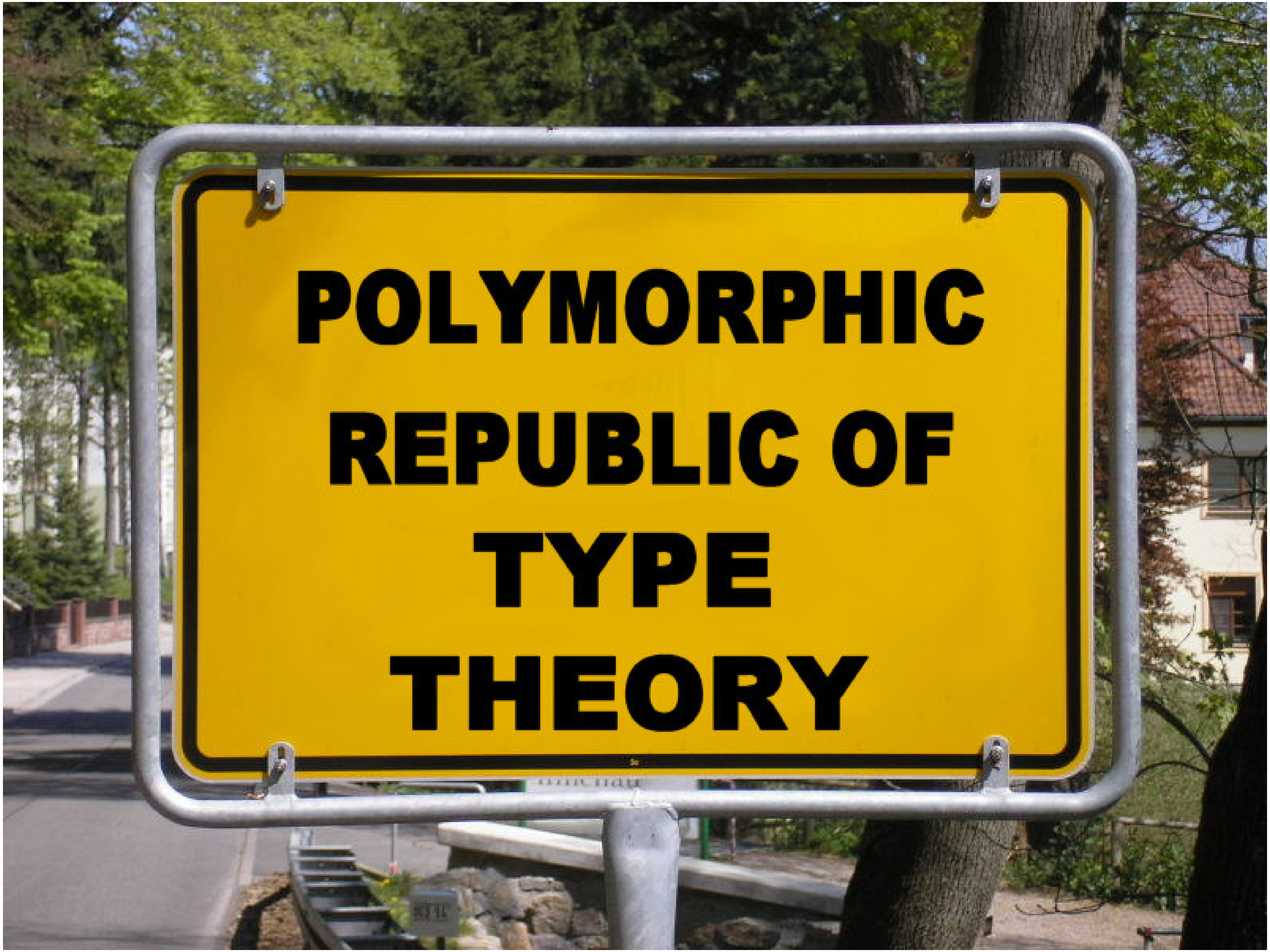
```
   (if x + 5 then 42 else 12, {x→3})
→ (if 3 + 5 then 42 else 12, {x→0})
→ (if 8 then 42 else 12, {x→0})
```

## 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

$$42 : int$$

$$1 + 5 : int$$

$$true : bool$$

- Notation exists for more complex types; I'll only detail functional types

## Functional Types

- Functional types (that is, types of functions) use an arrow notation
  - The type of the arguments go to the left of the arrow
  - The type of the return value goes to the right of the arrow

$$sub\ double\ (int\ \$x)\ \{\ 2\ *\ \$x\ \}\ :\ int \rightarrow int$$

$$sub\ iszero\ (int\ \$x)\ \{\ \$x\ ==\ 0\ \}\ :\ int \rightarrow bool$$

## 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:

$$2 * \$x : int$$

- Formally we write this as follows:

$$\Gamma \vdash 2 * \$x : int$$

- 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

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

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

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

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

## 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:

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

- 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 $\Gamma$, 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!

## Type Inference

- Given a type environment and a term, type inference finds the type that the term has, if it does indeed have one.

Given a type environment $\Gamma$ and a term $t$, find a type $T$ such that $\Gamma \vdash t : T$

- Often seen in functional languages (ML, Haskell).
- Computationally harder than type checking; type inference problem is undecidable for some type systems!

## Type Safety

- Type systems provide a way to ensure that our programs cannot perform certain bad operations at runtime
- For example, most high level languages only allow a reference to be used in a de-reference operations.
- Not the case in all languages; in C can create a pointer from any integer => programs can segfault

# Type Safety

- Perl 5's type system only allows references to be de-referenced; you get a runtime "type error" if you try to de-reference an integer (with strict on)

```
$ cat test.pl
#!/usr/bin/perl
use strict;
my $bar = 0xdeadbeef;
print $$bar;
$ perl test.pl
Can't use string ("3735928559") as a SCALAR ref while
"strict refs" in use at test.pl line 4.
```

## Type Safety

- Compare that with what C's type system lets you do

```
int main()
{
    int x = 0xdeadbeef;
    int* p = (int*)x; /* int becomes int pointer! */
    int y = *p; /* Dereference...KABOOM! */
    return 0;
}
```

- This program will produce a segfault when you run it

## Type Safety

- The distinction we are making here is that Perl is **type safe**, while C is not
- Type safety is a (highly desirable) property of the type system, but for any complex type system, it is not usually obvious that it is type safe
- If we formally describe the type system with induction rules, we can prove type safety!

## Static vs. Dynamic Typing

- The distinction being made is when type checking takes place
- Statically typed languages will type check the entire program at compile time
- Dynamically typed languages usually require values to carry their types around with them and perform a check at runtime when a value is used

## Static vs. Dynamic Typing Example

- The following program may work fine in a dynamically typed language, but fail to compile under a statically typed one

```
x = “foo”
if (complex condition that is always true)
    x = 39
y = x + 3
```

- Value always an integer by the time x is used in the add operation; static type check can’t determine this

# Strong vs. Weak Typing

- A vague definition: "how strictly are type rules enforced?"
- A strongly typed language (e.g. C#) would reject the following program; a weakly typed language (Visual Basic, Perl) would accept it

```
x = 42;
y = "20"
z = x + y
```

## Strong vs. Weak Typing

- Strongly typed languages generally enforce that coercions between types that may cause data loss (such as string to integer) must be written explicitly as casts
- Weakly typed languages assume the programmer knows what they are doing (not always a good assumption!) and performs a coercion implicitly

## 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
- I will quickly look at three of them that feature in Perl 6 in some form
  - Subclassing
  - Parametric polymorphism (aka generics and parameterised types)
  - Refinement types

## Subclassing

- More commonly known as inheritance
- A key part of object oriented programming
- A subclass may be used in place of a parent class because it only adds to the behaviours and representation that the parent class has
- Found in the many OO languages

## Subclassing

- 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")

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

## Parametric Polymorphism

- Key idea: a type can take type parameters, just as a function takes function parameters
- We could define the types “integer list”, “string list”, etc.
- Parametric polymorphism allows us to give the list the type “α list”, where α is a type parameter that we supply when using the list

# Parametric Polymorphism

- For example, we could implement a parametric List type in C# 2.0 that looks something like this:

```
public class List<T>
{
    public void Add(T value)
    {
        ...
    }
    public T Get(int index)
    {
        ...
    }
```

## Parametric Polymorphism

- The type parameter is supplied when an instance of the list class is created

```
List<int> = new List<int>();
```

- Perl 6 provides parametric polymorphism in an interesting way!
- A role (basically a group of methods that are composed into a class) is implicitly parameterised on the type of the invocant

# 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

- Anonymous refinement types in Perl 6 will be very useful!

```
sub Halve (Int $n where { $^n % 2 == 0 }) returns Int
{
    return $n / 2;
}
```

- Can use a more refined type in place of a less refined one, providing yet another path to polymorphic code!

# The End

# Any questions?