The Latro Programming Language

So why are you doing this?

So why are you doing this?

- I love programming languages

So why are you doing this?

- I love programming languages
- The "state of the art" leaves plenty to be desired

So why are you doing this?

- I love programming languages
- The "state of the art" leaves plenty to be desired
- Learning

Why Existing Languages Are Not Good Enough

Why Existing Languages Are Not Good Enough

- Semantic baggage

Why Existing Languages Are Not Good Enough

- Semantic baggage
- Tooling/library baggage
  - OCaml: 3 standard libraries, 4 build systems

Why Existing Languages Are Not Good Enough

- Semantic baggage
- Tooling/library baggage
  - OCaml: 3 standard libraries, 4 build systems
- Inherited baggage from piggybacked ecosystems

Why Existing Languages Are Not Good Enough

- Semantic baggage
- Tooling/library baggage
  - OCaml: 3 standard libraries, 4 build systems
- Inherited baggage from piggybacked ecosystems
- Ugly syntax

Why Existing Languages Are Not Good Enough

- Semantic baggage
- Tooling/library baggage
  - OCaml: 3 standard libraries, 4 build systems
- Inherited baggage from piggybacked ecosystems
- Ugly syntax
- Too many features

Why Existing Languages Are Not Good Enough

- Semantic baggage
- Tooling/library baggage
  - OCaml: 3 standard libraries, 4 build systems
- Inherited baggage from piggybacked ecosystems
- Ugly syntax
- Too many features
- Too few features

Why Existing Languages Are Good Enough

Why Existing Languages Are Good Enough

- Too many already

Why Existing Languages Are Good Enough

- Too many already
- Stable, mature, performant

Why Existing Languages Are Good Enough

- Too many already
- Stable, mature, performant
- Lots of libraries

Why Existing Languages Are Good Enough

- Too many already
- Stable, mature, performant
- Lots of libraries
- We're intimately familiar with their idiosyncracies and shortcomings

Why Existing Languages Are Good Enough

- Too many already
- Stable, mature, performant
- Lots of libraries
- We're intimately familiar with their idiosyncracies and shortcomings
- Interoperability

Why Existing Languages Are Good Enough

- Too many already
- Stable, mature, performant
- Lots of libraries
- We're intimately familiar with their idiosyncracies and shortcomings
- Interoperability
- Importance of things like syntax is overstated

How I Approach Design

How I Approach Design

- What is my ideal language?

How I Approach Design

- What is my ideal language?
- What do other people hate in their everyday languages? (see #talk-java)

How I Approach Design

- What is my ideal language?
- What do other people hate in their everyday languages? (see #talk-java)
- Writing fake programs

How I Approach Design

- What is my ideal language?
- What do other people hate in their everyday languages? (see #talk-java)
- Writing fake programs
- Cost vs. benefit

How I Approach Design

- What is my ideal language?
- What do other people hate in their everyday languages? (see #talk-java)
- Writing fake programs
- Cost vs. benefit
- Bottom up

What is Latro?

What is Latro?

- Functional

What is Latro?

- Functional
- Strict, call-by-value

What is Latro?

- Functional
- Strict, call-by-value
- Static typing, with type inference

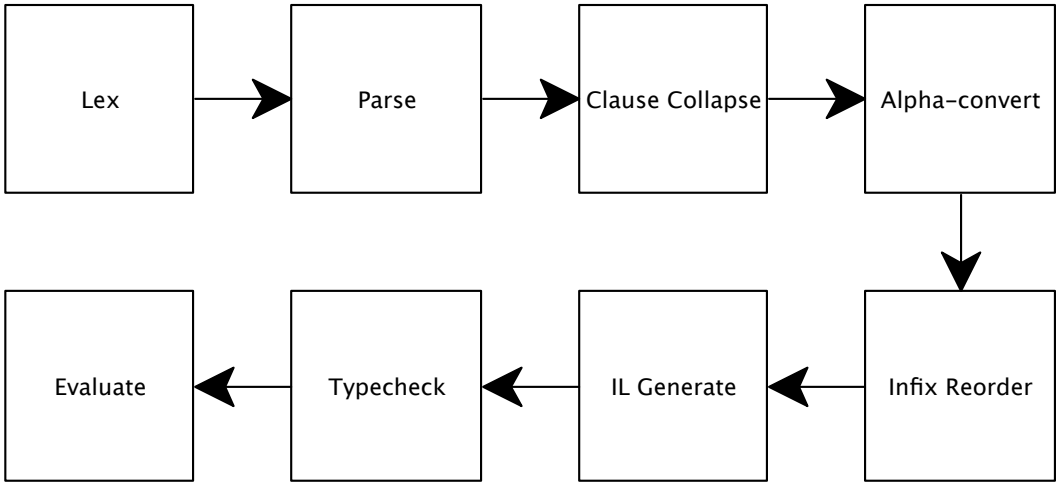
What is Latro?

- Functional
- Strict, call-by-value
- Static typing, with type inference
- Parametric polymorphism

What is Latro?

- Functional
- Strict, call-by-value
- Static typing, with type inference
- Parametric polymorphism
- Some side effects allowed (IO), but no mutation/rebinding

Compiler Phases



Lexing

- Translate source text into a stream of tokens

Lexing

- Translate source text into a stream of tokens

```
def x = 1 + 2 * 3
```

Lexing

- Translate source text into a stream of tokens

```
def x = 1 + 2 * 3
```

⇓

```
[KwDef, Id("x"), OpEq,
 NumLit("1"), Id("+"),
 NumLit("2"), Id("*"), NumLit("3")]
```

Parsing

- Translate token stream into a syntax tree

Parsing

- Translate token stream into a syntax tree

```
[KwDef, Id("x"), OpEq,
 NumLit("1"), Id("+"),
 NumLit("2"), Id("*"), NumLit("3")]
```

⇓

```
(ExpAssign
  (PatExpId "x")
  (ExpInfixApp
    (Id "*")
    (ExpInfixApp
      (Id "+")
      (ExpNumLit 1)
      (ExpNumLit 2))
    (ExpNumLit 3)))
```

Clause Collapsing

- Combine function clauses into single functions

Clause Collapsing

- Combine function clauses into single functions

```
and(True, True) = True
and(_, _) = False
```

Clause Collapsing

- Combine function clauses into single functions

```
and(True, True) = True
and(_, _) = False
```

⇓

```
and(a, b) {
  def args = %(a, b)
  switch (args) {
    case %(True, True) -> True
    case %(_, _)       -> False
    case _             -> fail("Match fail!")
  }
}
```

Alpha Conversion

- Rewrite the syntax tree such that all identifiers are unique

Alpha Conversion

- Rewrite the syntax tree such that all identifiers are unique
- Flatten modules into top-level sequences of bindings

Alpha Conversion

- Rewrite the syntax tree such that all identifiers are unique
- Flatten modules into top-level sequences of bindings
- Rewrite all qualified identifiers as simple ones

Alpha Conversion

- Two passes

Alpha Conversion

- Two passes
  - Build an environment mapping raw id's to either unique id's or sub-environments

Alpha Conversion

- Two passes
  - Build an environment mapping raw id's to either unique id's or sub-environments
  - Rewrite all raw id references into unique-id references (using the environment)

Alpha Conversion

```
module Foo {
   x = 42
   y = x
   z = Bar.x
}
module Bar {
   x = 43
   y = x
}
```

```
x@1 = 42
y@2 = x@1
z@3 = x@4

x@4 = 43
y@5 = x@4
```

Infix Reordering

- Rewrite infix applications according to user-defined precedence levels

Infix Reordering

```
1 + 2 * 3 - 4 / 5
```

Infix Reordering

```
1 + 2 * 3 - 4 / 5
```

⇓

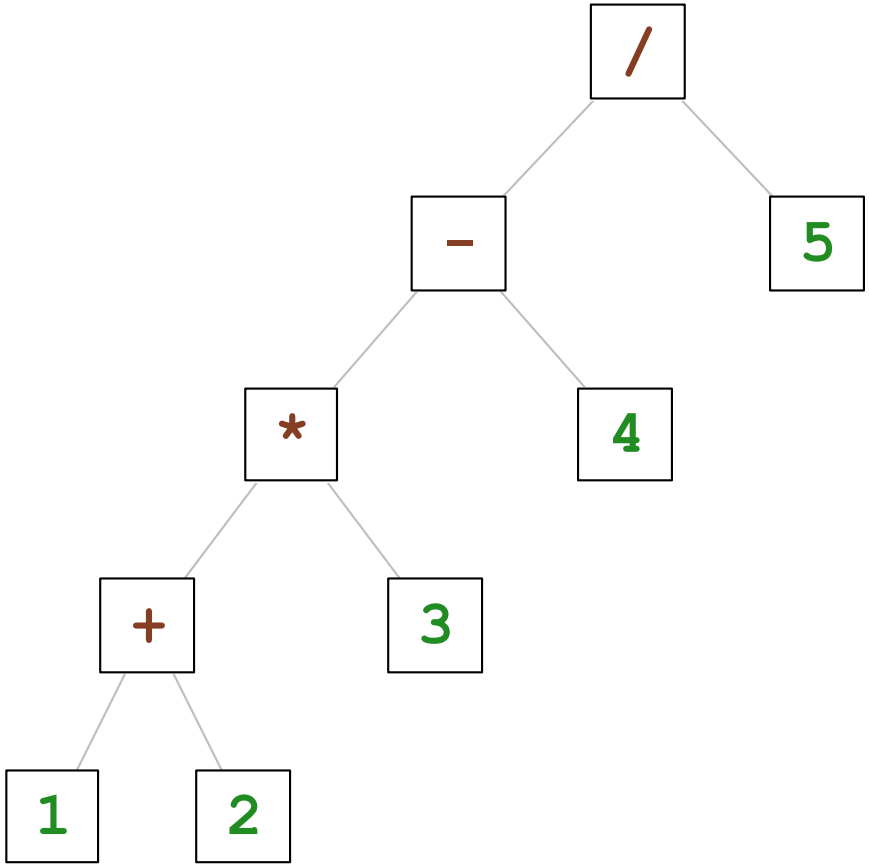
```
(((1 + 2) * 3) - 4) / 5
```

Infix Reordering

```
(((1 + 2) * 3) - 4) / 5
```

Infix Reordering

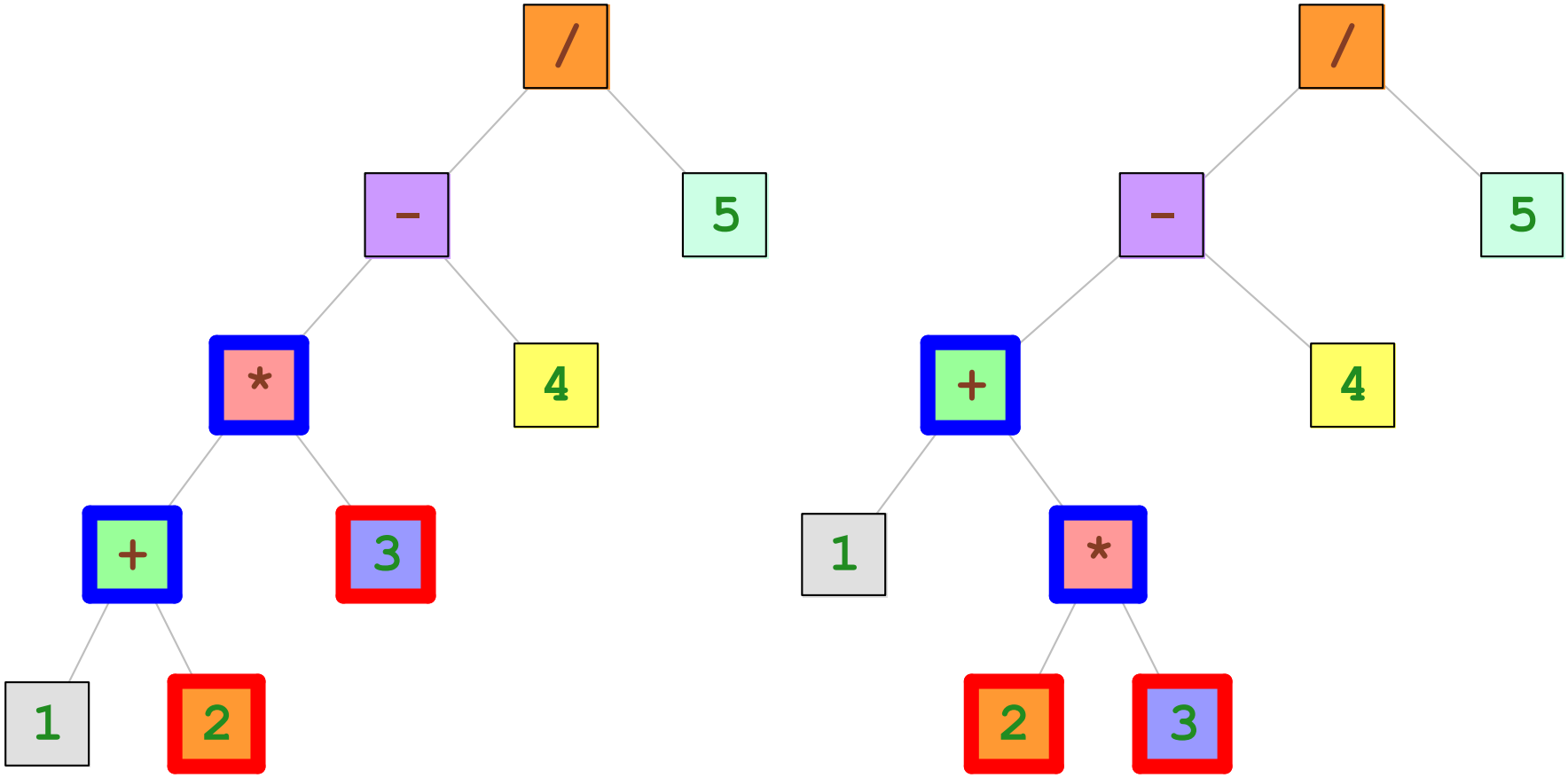
```
(((1 + 2) * 3) - 4) / 5
```



Infix Reordering

```
(((1 + 2) * 3) - 4) / 5
```

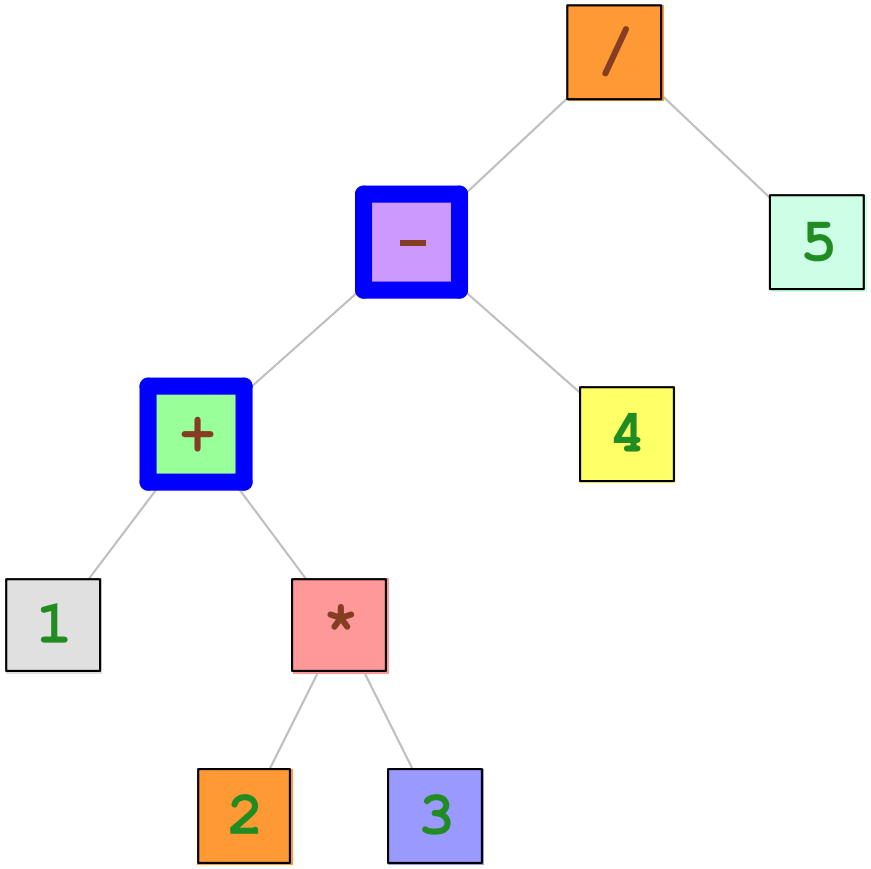
```
precedence(*) > precedence(+)
```

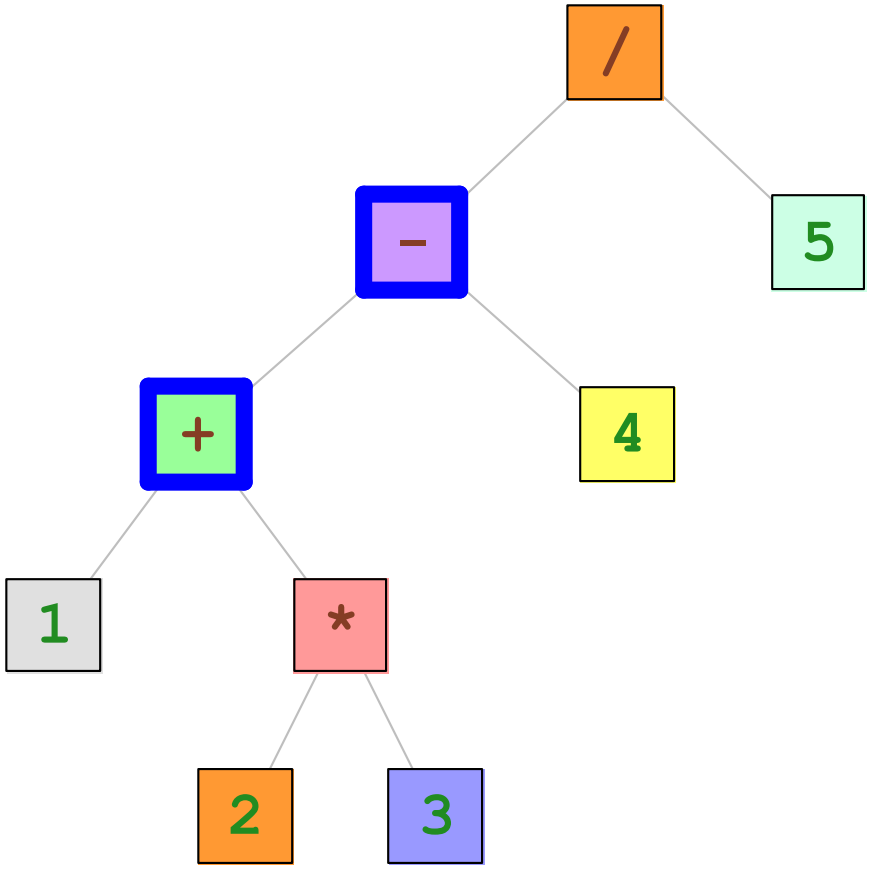


Infix Reordering

```
((1 + (2 * 3)) - 4) / 5
```

```
precedence(+) == precedence(-)
```

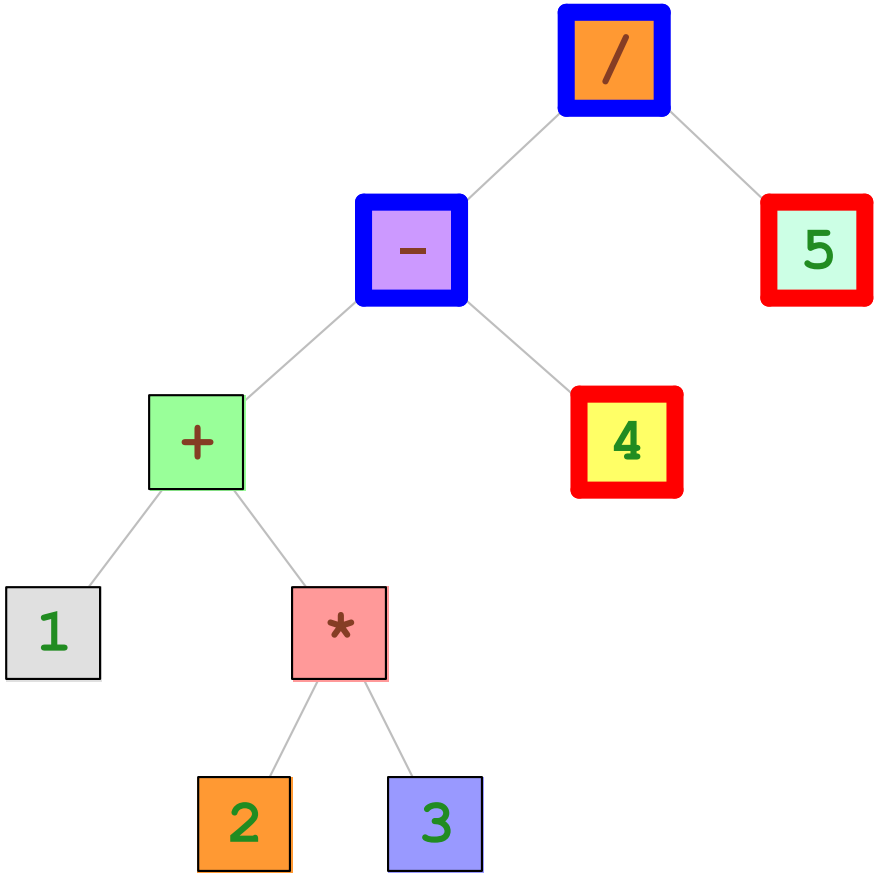


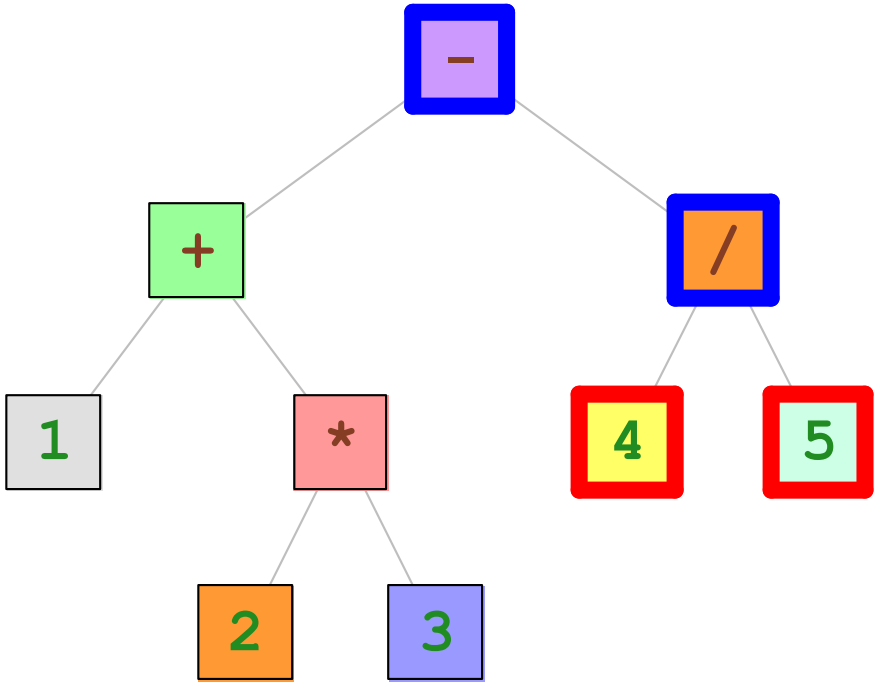


Infix Reordering

```
((1 + (2 * 3)) - 4) / 5
```

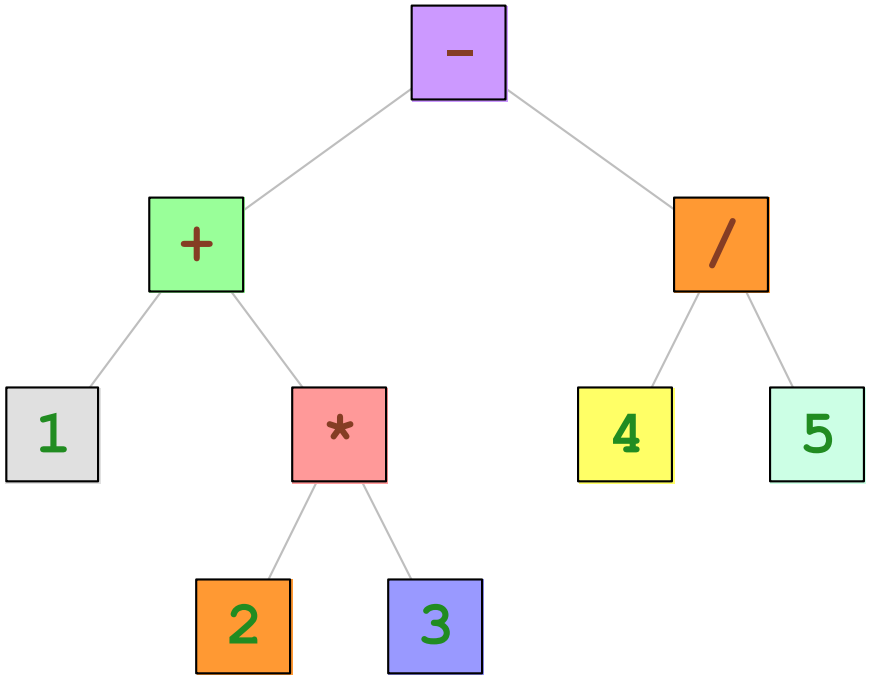
```
precedence(/) > precedence(-)
```





Infix Reordering

```
(1 + (2 * 3)) - (4 / 5)
```



IL Generation

- Translate high-level AST into slightly-lower-level AST

```
(ExpAssign
  (PatExpId "x@1")
  (ExpInfixApp
    (Id "+")
    (ExpNumLit 1)
    (ExpNumLit 2)))
```

IL Generation

- Translate high-level AST into slightly-lower-level AST

```
(ExpAssign
  (PatExpId "x@1")
  (ExpInfixApp
    (Id "+")
    (ExpNumLit 1)
    (ExpNumLit 2)))
```

⇓

```
(ILAssign
  "x@1"
  (ILApp
    (ILRef "+")
    (ILNumLit 1)
    (ILNumLit 2)))
```

Typechecking

- Verify that the program is well-typed

Typechecking

- Verify that the program is well-typed
- Annotate the syntax tree with type information

Typechecking

```
(ILAssign
  "x@1"
  (ILApp
    (ILRef "+")
    (ILNumLit 1)
    (ILNumLit 2)))
```

⇓

```
(ILAssign
  TyUnit
  "x@1"
  (ILApp
    TyInt
    (ILRef (TyArrow (TyInt TyInt TyInt)) "+")
    (ILNumLit TyInt 1)
    (ILNumLit TyInt 2)))
```

Evaluation

- Run the program!

Evaluation

- Run the program!

```
(ILApp
  TyInt
  (ILRef (TyArrow (TyInt TyInt TyInt)) "+")
  (ILNumLit TyInt 1)
  (ILNumLit TyInt 2))
```

⇓

```
(ValueInt 3)
```

Thanks!

- Special thanks to Ayo and Drew
- https://github.com/Zoetermeer/latro
- https://github.com/Zoetermeer/latro/tree/master/papers
- https://github.com/Zoetermeer/latro/blob/master/talks/bt-products.rkt