CSCI 275: Programming Abstractions

Lecture 33: Learning a Language (cont.)

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Reminders about the Lambda Calculus

The content in these lectures is adapted from “Types and Programming Languages” by Pierce, Cornell’s CS 4110/6110 Notes and Phil Wadler’s ”Propositions as Types”

Why learn the Lambda Calculus?

We learn a lot of fundamentals as part of the CS major:

- CSCI 151/280
    - Runtime analysis
    - P = NP
- CSCI 383
    - Computability
    - Decideability
    - Turing Machines

Why learn the Lambda Calculus?

We learn a lot of fundamentals as part of the CS major:

- CSCI 151/280
    - Runtime analysis
    - P = NP
- CSCI 383
    - Computability
    - Decideability
    - Turing Machines

The lambda calculus provide another way to think about the mathematical foundations.

In particular, there is a direct equivalence between the Lambda Calculus & Turing Machines

What is the lambda calculus?

The lambda calculus' "importance arises from the fact that it can be viewed simultaneously as as a simple programming language *in which* computations can be described and as a mathematical object *about which* rigorous statements can be proved." (Pierce)

Other Foundational Formalisms

Pi-calculus: the core language for defining concurrent programming languages

Object calculus: the core language for defining object-oriented languages

The Lambda Calculus

Much like other languages, the lambda calculus has a *syntax* and a *semantics*. Here is its syntax:

$e ::=$	x	*variable*
	$\lambda x.\ e$	*function abstraction*
	$e_1\ e_2$	*function application*

How do we compute with this?

It is *very simple*: all we can do in the base lambda calculus is apply functions to arguments.

Examples:

`(λx. x) a` gives `a`

`(λx. x (λx. x)) b` gives us `b (λx. x)`

How do we compute with this?

It is *very simple*: all we can do in the base lambda calculus is apply functions to arguments.

Rewriting these rules is called *beta-reduction*

Examples:

`(λx. x) a` gives `a`

`(λx. x (λx. x)) b` gives us `b (λx. x)`

These terms are called *reducible expressions*

Some Encodings

```
 and = λb. λc. b c false
true = λt. λf. t
false = λt. λf. F
zero = λf. λx. x
 one = λf. λx. f x
succ = λn. λf. λx. f (n f x)
```

Languages have different evaluation strategies

Formal beta-reduction rule

Formally the semantic rule is

$$(\lambda x.\ e)\ e_1 \rightarrow e\ \{e_1/x\}$$

In English we describe this as "the term obtained by replacing all free occurrences of x in e by e_1"

There are different ways to do beta-reduction!

It all is dependent on *which* reducible expressions you are allowed to reduce.

These are typically called ***evaluation strategies***

Let’s think about the following more complex reducible expression:

`(λx. x) ((λx. x) (λz. (λx. x) z))`

If we want to simplify the below expression and replace all instances of the "identity" procedure `(λx. x)` with the term `id`, what do we get?

```
(λx. x) ((λx. x) (λz. (λx. x) z))
```

A. `id (λz. id z)`

B. `id (id (λz. id z))`

C. `id (id (λz. z))`

D. `(λz. z)`

E. Something else

Full Beta-Reduction: Reduce Any Term!

Under full beta-reduction we can reduce in *any* order we want:

```
      id (id (λz. id z))
   -> id (λz. id z))
   -> (λz. id z))
   -> λz. z
```

Remember id is the identity procedure
`λx. x`

Normal Order: Leftmost, Outmost

Under normal order we start with the leftmost, ***outermost*** reducible expression:

```
    id (id (λz. id z))
 -> id (λz. id z)
 -> λz. id z
 -> λz. z
```

Applicative Order: Leftmost, Innermost

Under applicative order we start with the leftmost, ***innermost*** reducible expression:

```
      id (id (λz. id z))
->    id (id (λz. z))
->    id (λz. z)
->    λz. z
```

We typically do not evaluate *inside* lambdas

In most languages, we will not do the **`id z`** reductions below.

Normal Order

```
    id (id (λz. id z))
->  id (λz. id z)
->  λz. id z
->  λz. z
```

Applicative Order

```
    id (id (λz. id z))
->  id (id (λz. z))
->  id (λz. z)
->  λz. z
```

We typically do not evaluate *inside* lambdas

In Racket, when we define a `lambda` expression, we **do not evaluate its body:**

```
(lambda (x)
         (displayln "banana"))
```

`“banana”` does not print out.

Think about how we evaluate lambdas in MiniScheme

Call-by-Name Reduction

Normal order (outermost), but we do not reduce inside the bodies of λ-abstractions:

```
       id (id (λz. id z))
    -> id (λz. id z)
    -> λz. id z
```

Call-by-Value Reduction

Applicative order (innermost), but we do not reduce inside the bodies of λ-abstractions:

```
     id (id (λz. id z))
  -> id (λz. id z)
  -> λz. id z
```

We’ve seen CBN/CBV before!

This is the *formal* model of call-by-value, we discussed the way it is (or could be) implemented in Racket as parameter passing styles

Call by Name Example in Racket

```
(let* ([v 0]
       [f (lambda (x)
            (set! v (+ v 1))
            x)])
  (f (+ v 5)))
```

The text of `f`'s body becomes the two expressions (by replacing `x` with the text of the argument)

```
(set! v (+ v 1))
(+ v 5)
```

`v` is set to 1 and then 6 is returned

Call by Value Example in Racket

```
(let* ([v 0]
       [f (lambda (x)
            (set! v (+ v 1))
            x)])
  (f (+ v 5)))
```

`f` is called with value 5, so `x` is bound to 5

`v` is set to 1

`x` equal to 5 is returned

Abstract versus Concrete Syntax

Abstract/Concrete Syntax

Concrete Syntax: the characters that programmers actually write to create the language

MiniScheme expressions you wrote in minischeme.rkt REPL

Abstract Syntax: the internal representation of programs as labeled trees

What you created in parse.rkt!

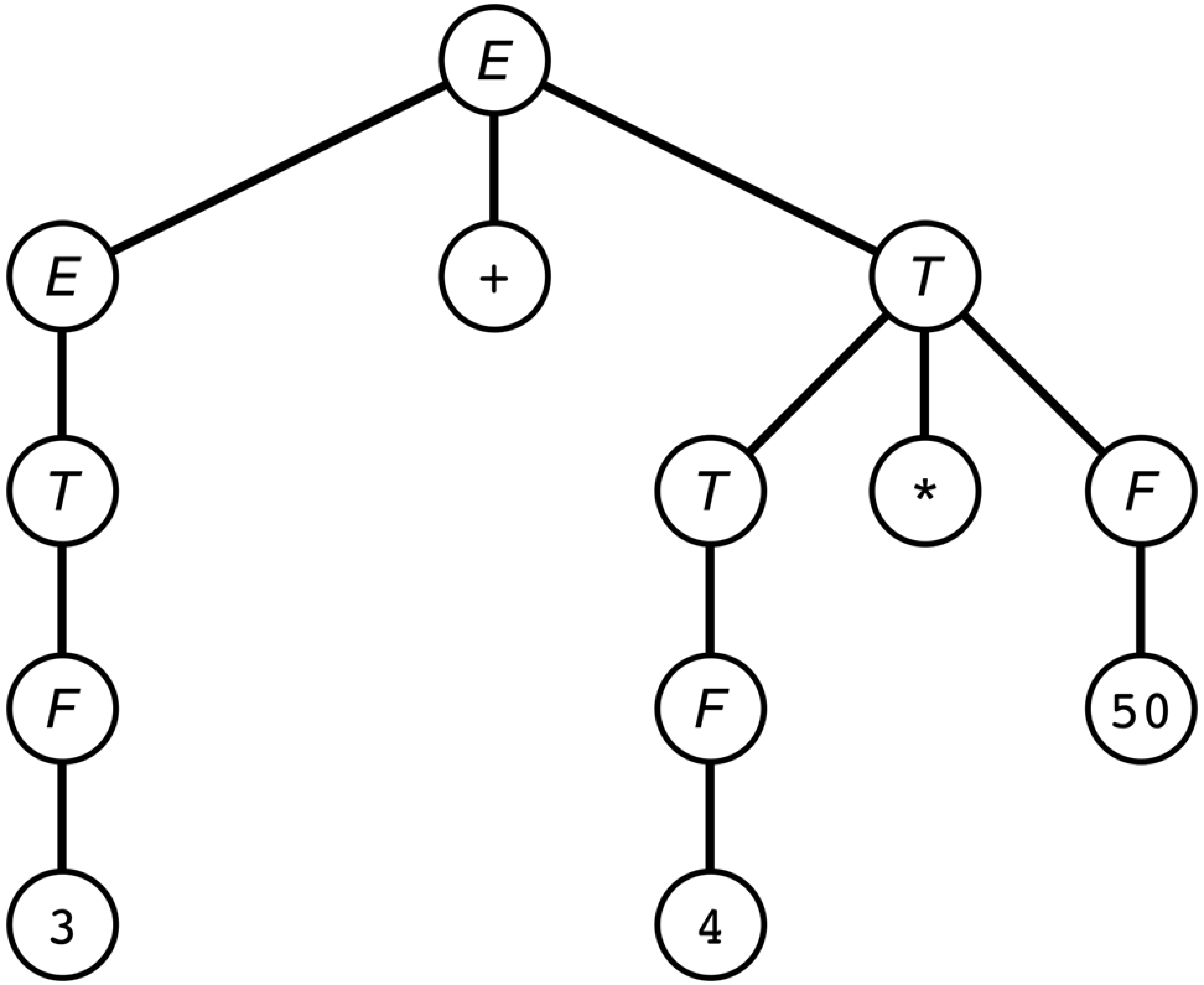
Lambda Calculus Provides Abstract Syntax

As Pierce states, “Grammas like the one for lambda-terms above should be understood as describing legal tree structures, not strings of tokens or characters”

Lambda terms are guidelines for an *abstract* representation of a computation that can be instantiated in many ways

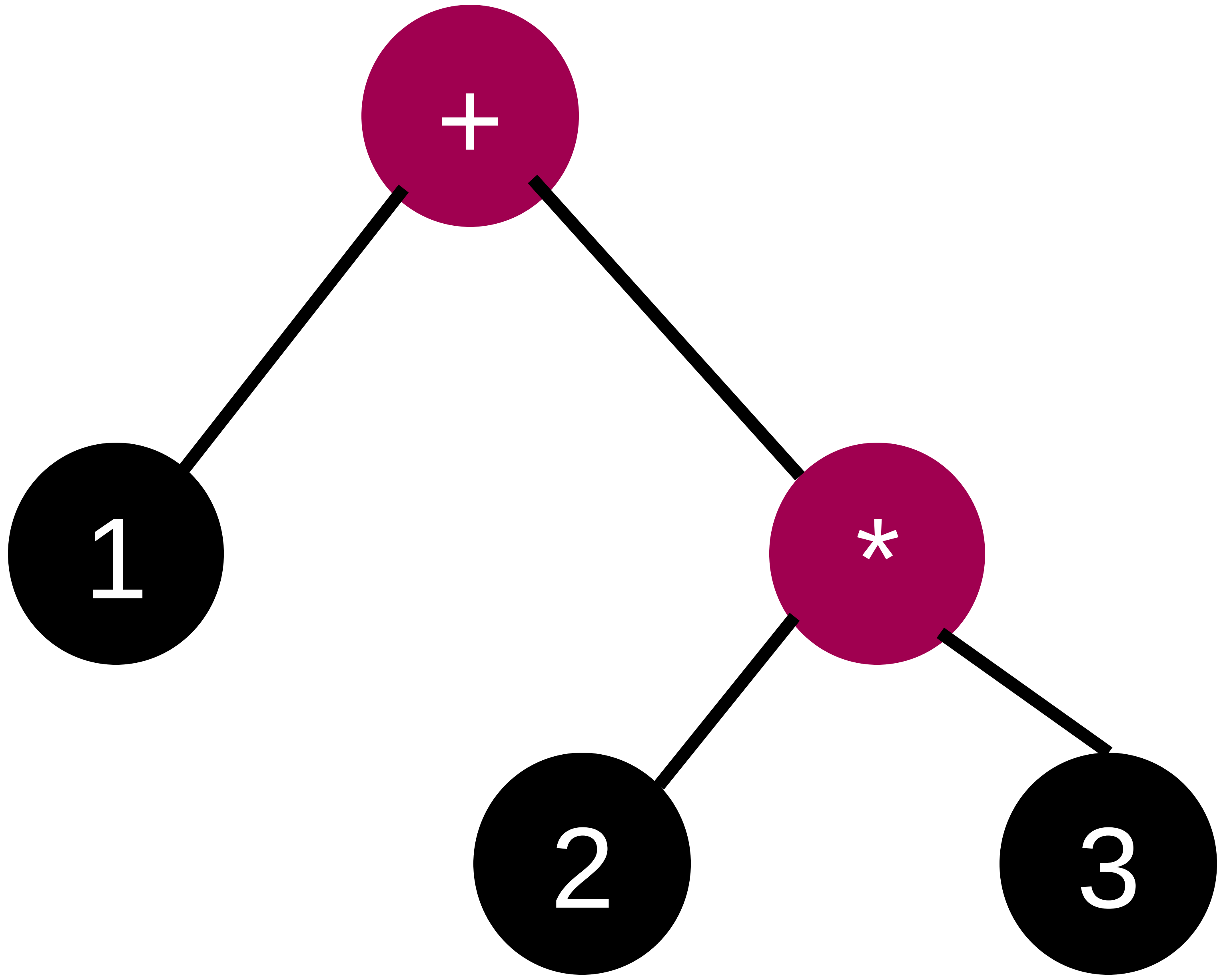
Parse Trees & Abstract Syntax Trees

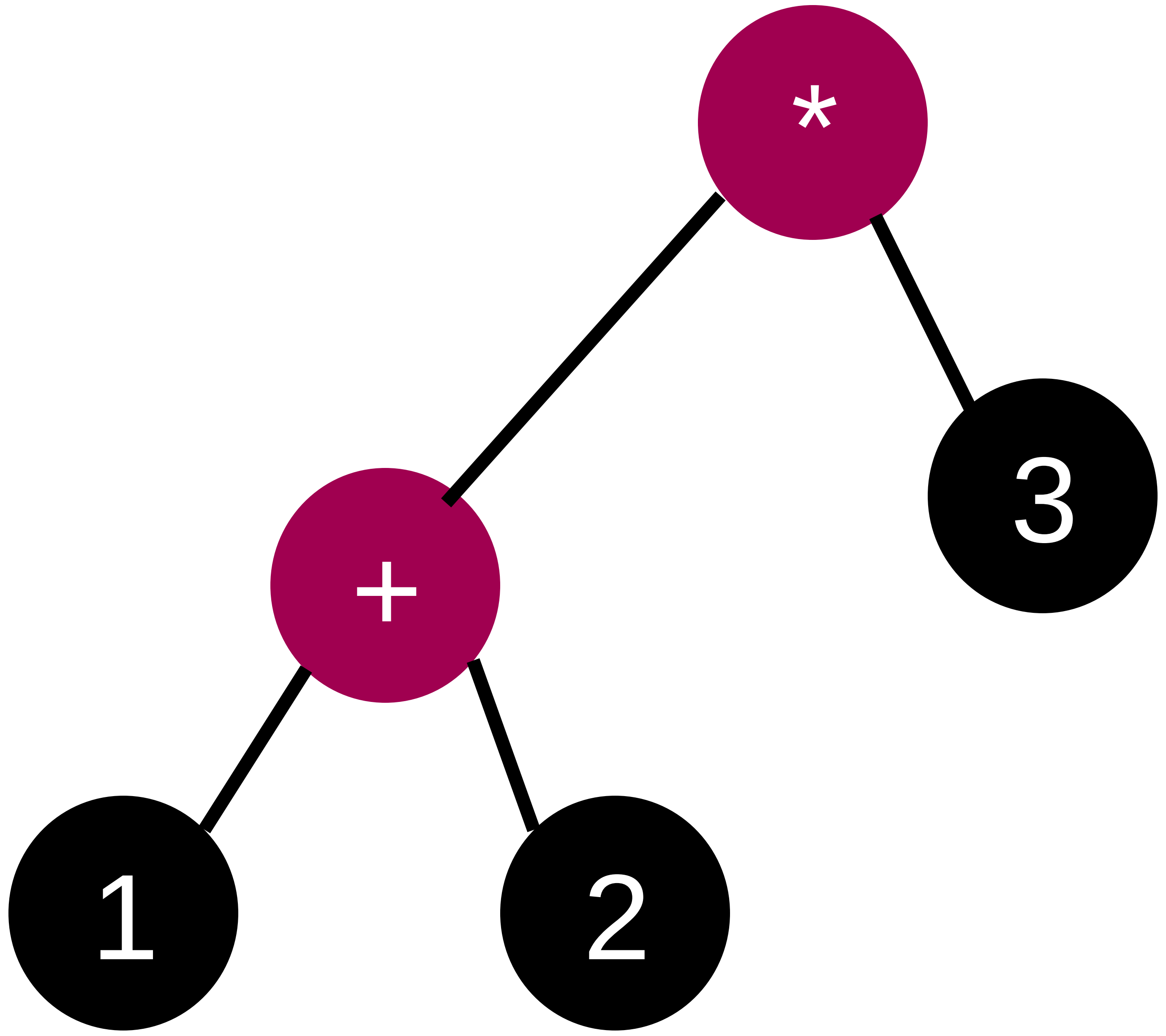
Parsers (like the one you wrote in MiniScheme) take a sequence of tokens and *create* an abstract syntax tree from them



Abstract Syntax Trees

ASTs can easily encode precedence operations–consider `1 + 2 * 3`





Consider the following two expressions:

Python: `1 + 2 - 3 * 4`

Racket: `(+ 1 (- 2 (* 3 4)))`

Which of the following statements do you agree with?

A. Easier to determine the order of precedence in Racket than Python

B. Easier to determine how to parse Racket than Python

C. Easier to determine the order of precedence in Python than Racket

D. Easier to determine how to parse Python than Racket

E. More than one of the above

Concrete & Abstract Syntax Similarity

In Scheme/Racket there is a *closeness* between the concrete syntax (what we write) and the abstract syntax

The language would *still work* without the closeness, but MiniScheme would likely have been harder to implement!