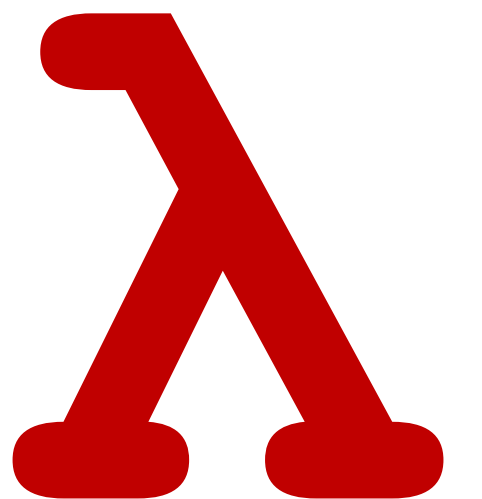


CSCI 275: Programming Abstractions

**Lecture 33: Learning a Language (cont.)
Spring 2025**

**Stephen Checkoway
Slides from Molly Q Feldman**




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 complex reducible expression:

$$(\lambda x. x) \ ((\lambda x. x) \ (\lambda z. (\lambda x. x) \ z))$$

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

$(\lambda x. x) ((\lambda x. x) (\lambda z. (\lambda x. x) z))$

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:

$\text{id } (\text{id } (\lambda z. \text{id } z))$
→ $\text{id } (\lambda z. \text{id } z)$
→ $(\lambda z. \text{id } z)$
→ $\lambda z. z$

Remember id is the
identity procedure

$\lambda x. x$

Normal Order: Leftmost, Outermost

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 ( $\lambda z.$  id z) )
```

```
-> id ( $\lambda z.$  id z)
```

```
->  $\lambda z.$  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))`

\rightarrow `id (λz. id z)`

\rightarrow `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

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`

Call-by-Name

`id (id (λz. id z))`
→ `id (λz. id z)`
→ `λz. id z`

Call-by-Value

`id (id (λz. id z))`
→ `id (λz. id z)`
→ `λz. id z`

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!

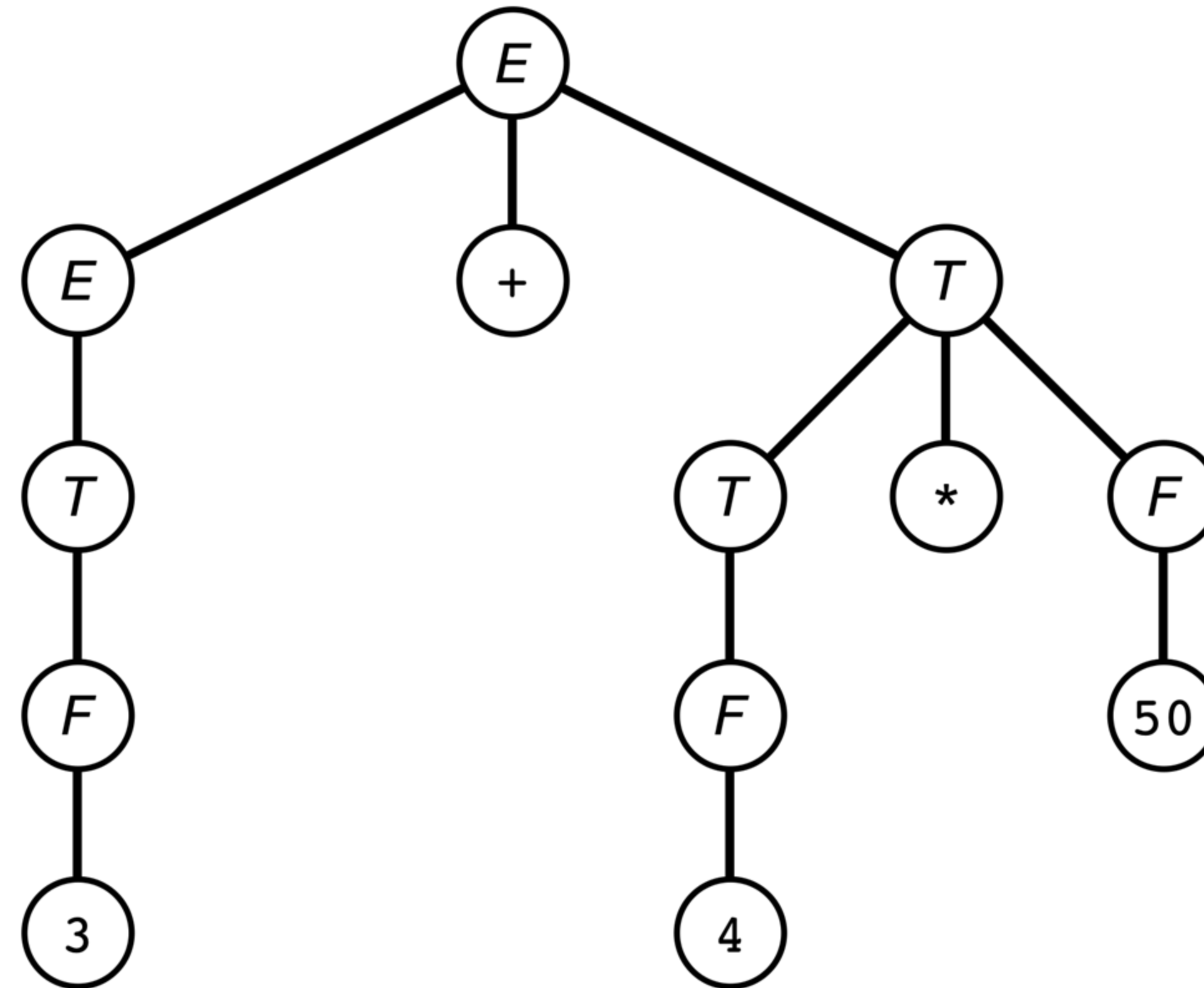
Lambda Calculus Provides Abstract Syntax

As Pierce states, “Grammars 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

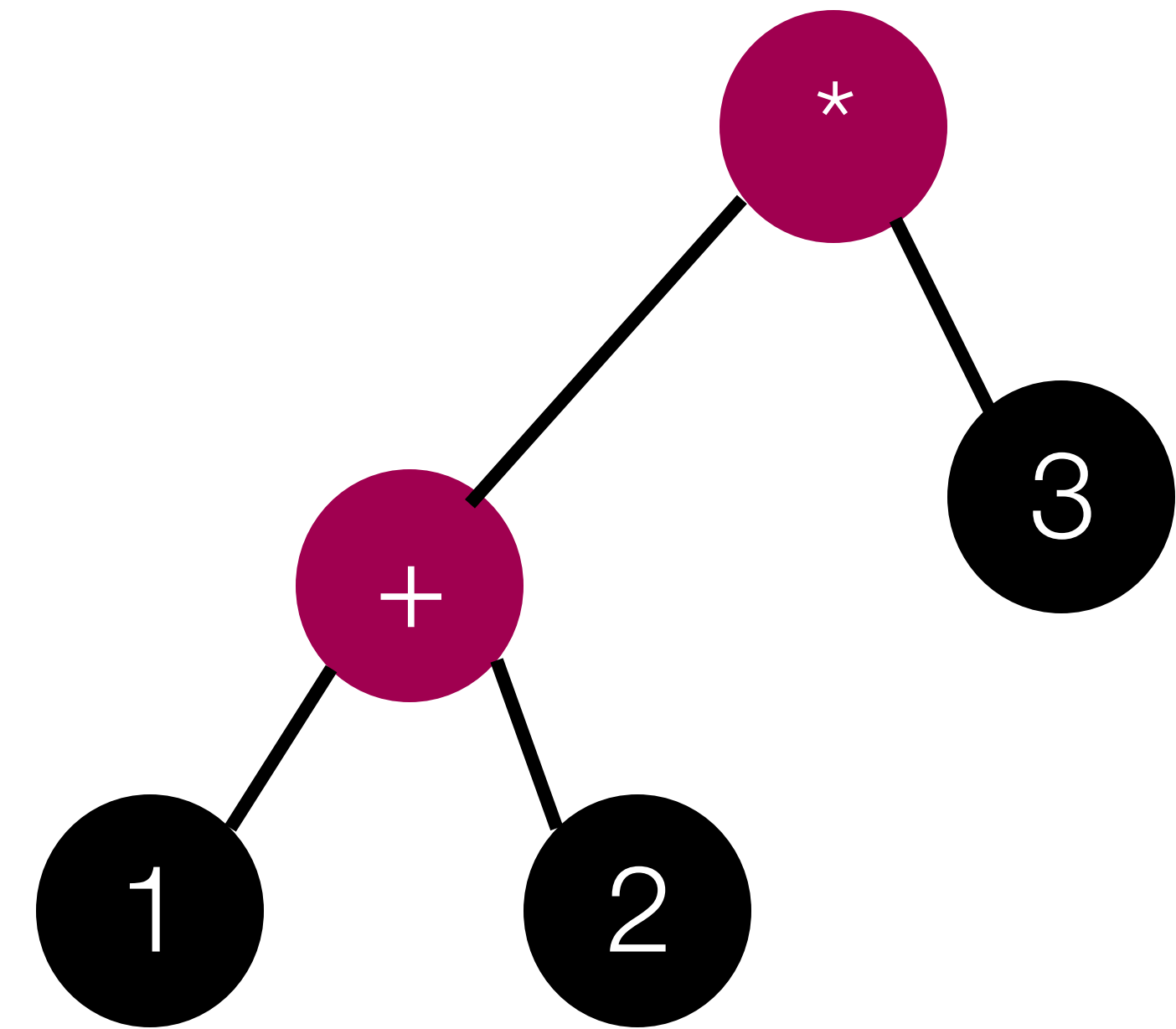
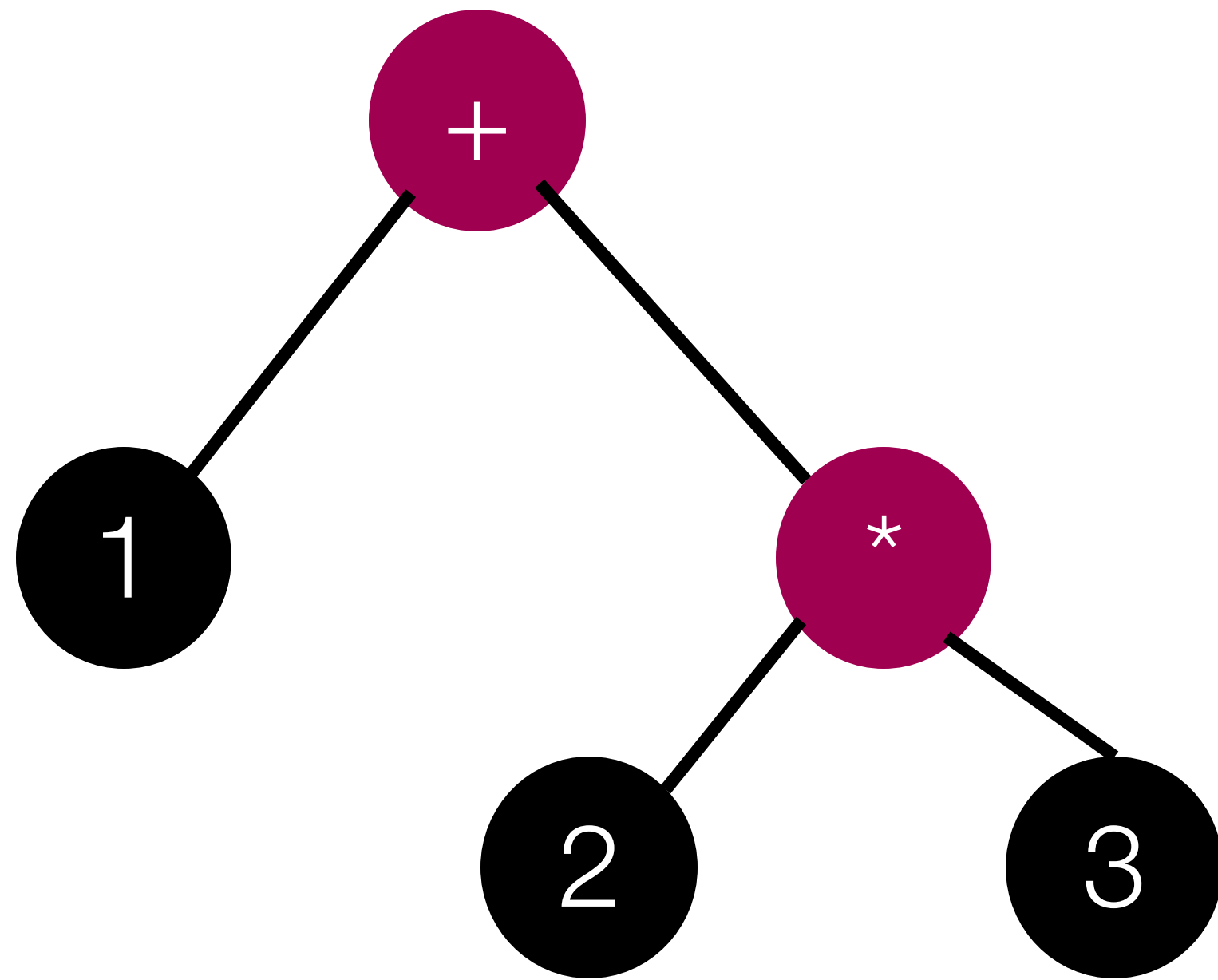
Parse Trees & Abstract Syntax Trees

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!