Programming Abstractions

Week 7-1: MiniScheme Interpreter

Stephen Checkoway

Project overview

In the next few homeworks, you'll write a small Scheme interpreter

The project has two primary functions

- `(parse exp)` creates a tree structure that represents the expression `exp`
- `(eval-exp tree environment)` evaluates the given expression `tree` within the given `environment` and returns its value

We need a way to represent environments and we need some way to manipulate them

Environments

Environments are used repeatedly in `eval-exp` to look up the value bound to a symbol

There are two functionalities we need with environments

The first is we want to look up the value bound to a symbol; e.g.,

```
(let ([x 3])
  (let ([x 4])
    (+ x 5)))
```

should return 9 since the innermost binding of x is 4

Environments

Second, we need to produce new environments by extending existing ones

```
(let ([x 3])
  (+ (let ([x 10])
       (* 2 x))
     x))
```

evaluates to 23

- If `E0` is the top-level environment, then the first let extends `E0` with a binding of `x` to 3
- If `E1` is the new environment, we write `E1 = E0[x ↦ 3]`
- The second let creates a new environment `E2 = E1[x ↦ 10]`
- The `(* 2 x)` is evaluated using `E2`
- The final `x` is evaluated using `E1`

Let `E0` be an environment with `x` bound to 10 and `y` bound to 23.

Let `E1 = E0[x ↦ 8, z ↦ 0]`

What is the result of looking up `x` in `E0` and `E1`?

A. `E0`: 10
   `E1`: 10

B. `E0`: 8
   `E1`: 8

C. `E0`: 10
   `E1`: 8

D. `E0`: 8
   `E1`: 10

E. `E1` can't exist because `z` isn't bound in `E0`

Let `E0` be an environment with `x` bound to 10 and `y` bound to 23.

Let `E1 = E0[x ↦ 8, z ↦ 0]`

What is the result of looking up `y` in `E0` and `E1`?

A. `E0`: 23
   `E1`: 23

B. `E0`: 23
   `E1`: error: `y` isn't bound in `E1`

C. It's an error in both because since `y` isn't bound in `E1`, it's not bound in `E0` any longer

D. None of the above

Let `E0` be an environment with `x` bound to 10 and `y` bound to 23.

Let `E1 = E0[x ↦ 8, z ↦ 0]`

What is the result of looking up `z` in `E0` and `E1`?

A. `E0`: 0
   `E1`: 0

B. `E0`: error: `z` isn't bound in `E0`
   `E1`: 0

C. None of the above

Extending environments

There are only two places where an environment is extended

Extending environments

Procedure call

The first is a procedure call

`(exp0 exp1 … expn)`

`exp0` should evaluate to a closure with three parts

- its parameter list;
- its body; and
- the environment in which it was created, i.e., the environment at the time the `(λ …)` that created the closure was evaluated

The other expressions are the arguments

The closure's environment needs to be extended with the parameters bound to the arguments

Extending environments

Procedure call

For example imagine the parameter list was `'(x y z)` and the arguments evaluated to `2`, `8`, and `'(1 2)`

If `E` is the closure's environment, then the closure's body should be evaluated with the environment

`E[x ↦ 2, y ↦ 8, z ↦ '(1 2)]`

Extending environments

Let expressions

The other situation where we extend an environment is a let expression

Consider

```
(let ([x (+ 3 4)]
      [y 5]
      [z (foo 8)])
  body)
```

We have three symbols `x`, `y`, and `z` and three values, 7, 5, and whatever the result of `(foo 8)` is, let's say it's 12

If E is the environment of the whole let expression then the body should be evaluated in the environment `E[x ↦ 7, y ↦ 5, z ↦ 12]`

Extending environments

In both cases we have

- A list of symbols
- A list of values
- A previous environment we're extending

This suggests a way to make an environment data type as a list:

```
('env syms vals previous-env)
```

and a constructor

```
(define (env syms vals previous-env)
  (list 'env syms vals previous-env))
```

Environment data type

Constructor for extending an environment (some error checking omitted)

```
(define (env syms vals previous-env)
  (list 'env syms vals previous-env))
```

The top-level environment doesn't have a previous environment so let's use model it as extending an empty environment

```
(define empty-env null)
```

The top-level environment can now be

```
(define top-level-env
  (env syms vals empty-env))
```

Looking up a binding

`(env-lookup environment symbol)`

Looking up `x` in an environment has two cases

If the environment is empty, then we know `x` isn't bound there so it's an error

Otherwise we look in the list of symbols of an extended environment

- If the symbol `x` appears in the list, then great, we have the value
- If the symbol `x` doesn't appear, then we lookup `x` in the previous environment

The main task of this first MiniScheme homework is to write `env-lookup`

We need some recognizers for our env

```
; Environment recognizers.
(define (env? e)
  (or (empty-env? e) (extended-env? e)))

(define (empty-env? e)
  (null? e))

(define (extended-env? e)
  (and (list? e)
       (not (null? e))
       (eq? (first e) 'env)))
```

We need a way to access the env fields

```
(define (env-syms e)
  (cond [(empty-env? e) empty]
        [(extended-env? e) (second e)]
        [else (error 'env-syms "e is not an env")]))

(define (env-vals e)
  (cond [(empty-env? e) empty]
        [(extended-env? e) (third e)]
        [else (error 'env-vals "e is not an env")]))

(define (env-previous e)
  (cond [(empty-env? e) (error 'env-previous "e has no previous env")]
        [(extended-env? e) (fourth e)]
        [else (error 'env-previous "e is not an env")]))
```

Alphabets and words

An alphabet Σ is a finite, nonempty set of symbols

- `{0, 1}` is a binary alphabet
- The set of emoji is an alphabet
- The set of English words is an alphabet

A word (also called a string) w over an alphabet Σ is a finite (possibly-empty) sequence of symbols from the alphabet

- The empty word, ε, consisting of no symbols is a word over every alphabet
- `001101` is a word over `{0, 1}`
- ⭐🐈🐈🐲💥🤦‍♀️ is a word over the emoji alphabet
- `functional programming is great` is a word over English

Languages

A language is a (possibly infinite) set of words over an alphabet

There's a whole lot we can do studying languages as mathematical objects

We're not going to do that in this course, take theory of computation to find out more!

For a given programming language (like Scheme) the alphabet is the set of keywords, identifiers, and symbols in the language

- This is a bit of a simplification because there are infinitely many possible identifiers but alphabets must be finite

A word (or string) over this alphabet is in the programming language if it is a syntactically valid program

Syntactically valid?

Consider the invalid Scheme program

```
(let ([x 5]
      [y 32])
  (+ z 2))
```

This is *syntactically* valid (i.e., it's a word in the Scheme language) but *semantically* meaningless as we don't have a binding for the identifier `z`

Grammars

A grammar for a language is a (mathematical) tool for specifying which words over the alphabet belong to the language

Grammars are often used to determine the meaning of words in the language

For example, consider the arithmetic expression a+b*c as a word over the alphabet consisting of variables and arithmetic operators

- We can write many different grammars that will let us determine if a given word is a valid expression (i.e., is in the language of valid expressions)
- With a careful choice of grammars we can determine that this means a+(b*c) and not (a+b)*c

Grammars are very old, dating back to at least Yāska the 4th c. BCE

Mathematical representation of grammars

A grammar G is a 4-tuple $G = (V, \Sigma, S, R)$ where

- V is a finite, nonempty set of *nonterminals*, also called variables
- Σ is an alphabet of *terminal* symbols
- $S \in$ V is the *start* nonterminal
- R is a finite set of *production rules*

(Terminal symbols are distinct from nonterminals)

In English, we might have nonterminals like *NOUN*, *VERB*, *NP*, etc.

We often write nonterminals in upper-case and terminals in lower-case

Production rules

Nonterminals are expanded using production rules to sequences of terminals and nonterminals

A production rule looks has the form
$A \to \alpha$

where A is a nonterminal and α is a (possibly-empty) word over $\Sigma \cup V$

Here's an example for Scheme
$EXP \to$ `(` `if` $EXP\ EXP\ EXP$ `)`

This says that wherever we have an expression, we can expand it to an if-then-else expression which starts with `(` followed by `if` and then three more expressions and lastly `)`

Example grammar for arithmetic

EXP → *EXP* + *TERM*
EXP → *TERM*
TERM → *TERM* ∗ *FACTOR*
TERM → *FACTOR*
FACTOR → （ *EXP* ）
FACTOR → number

Compact form：

EXP → *EXP* + *TERM* | *TERM*
TERM → *TERM* ∗ *FACTOR* | *FACTOR*
FACTOR → （ *EXP* ）| number

Derivations

A *derivation* with a grammar starts with a nonterminal and replaces nonterminals one at a time until only a sequence of terminals remains

A *left-most derivation* is a derivation where the nonterminal replaced in each step is the left-most nonterminal

Derivation example

Left-most derivation of 3 + 4 * 50

$EXP \Rightarrow$

$EXP \rightarrow EXP + TERM \mid TERM$
$TERM \rightarrow TERM * FACTOR \mid FACTOR$
$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of 3 + 4 * 50

$EXP \Rightarrow EXP + TERM$

$EXP \rightarrow EXP + TERM \mid TERM$
$TERM \rightarrow TERM * FACTOR \mid FACTOR$
$FACTOR \rightarrow (EXP) \mid \text{number}$

Derivation example

Left-most derivation of 3 + 4 * 50

$$
\begin{aligned}
EXP &\Rightarrow EXP + TERM \\
&\Rightarrow TERM + TERM
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&EXP \rightarrow EXP + TERM \mid TERM \\
&TERM \rightarrow TERM * FACTOR \mid FACTOR \\
&FACTOR \rightarrow (\ EXP\ ) \mid \text{number}
\end{aligned}
}
$$

Derivation example

Left-most derivation of 3 + 4 * 50

$$
\begin{aligned}
EXP &\Rightarrow EXP + TERM \\
&\Rightarrow TERM + TERM \\
&\Rightarrow FACTOR + TERM
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&EXP \rightarrow EXP + TERM \mid TERM \\
&TERM \rightarrow TERM * FACTOR \mid FACTOR \\
&FACTOR \rightarrow (\ EXP\ ) \mid \mathrm{number}
\end{aligned}
}
$$

Derivation example

Left-most derivation of 3 + 4 * 50

$$
\begin{aligned}
EXP &\Rightarrow EXP + TERM \\
&\Rightarrow TERM + TERM \\
&\Rightarrow FACTOR + TERM \\
&\Rightarrow 3 + TERM
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&EXP \rightarrow EXP + TERM \mid TERM \\
&TERM \rightarrow TERM * FACTOR \mid FACTOR \\
&FACTOR \rightarrow (\ EXP\ ) \mid \text{number}
\end{aligned}
}
$$

Derivation example

Left-most derivation of 3 + 4 * 50

$$
\begin{aligned}
EXP &\Rightarrow EXP + TERM \\
&\Rightarrow TERM + TERM \\
&\Rightarrow FACTOR + TERM \\
&\Rightarrow 3 + TERM \\
&\Rightarrow 3 + TERM * FACTOR
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&EXP \rightarrow EXP + TERM \mid TERM \\
&TERM \rightarrow TERM * FACTOR \mid FACTOR \\
&FACTOR \rightarrow (\ EXP\ ) \mid \text{number}
\end{aligned}
}
$$

Derivation example

Left-most derivation of 3 + 4 * 50

$$
\begin{aligned}
EXP &\Rightarrow EXP + TERM \\
&\Rightarrow TERM + TERM \\
&\Rightarrow FACTOR + TERM \\
&\Rightarrow 3 + TERM \\
&\Rightarrow 3 + TERM * FACTOR \\
&\Rightarrow 3 + FACTOR * FACTOR
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&EXP \rightarrow EXP + TERM \mid TERM \\
&TERM \rightarrow TERM * FACTOR \mid FACTOR \\
&FACTOR \rightarrow (\ EXP\ ) \mid \text{number}
\end{aligned}
}
$$

Derivation example

Left-most derivation of 3 + 4 * 50

$$\begin{aligned} EXP &\Rightarrow EXP + TERM \\ &\Rightarrow TERM + TERM \\ &\Rightarrow FACTOR + TERM \\ &\Rightarrow 3 + TERM \\ &\Rightarrow 3 + TERM * FACTOR \\ &\Rightarrow 3 + FACTOR * FACTOR \\ &\Rightarrow 3 + 4 * FACTOR \end{aligned}$$

$$\boxed{\begin{aligned} &EXP \rightarrow EXP + TERM \mid TERM \\ &TERM \rightarrow TERM * FACTOR \mid FACTOR \\ &FACTOR \rightarrow (\ EXP\ ) \mid \text{number} \end{aligned}}$$

Derivation example

Left-most derivation of 3 + 4 * 50

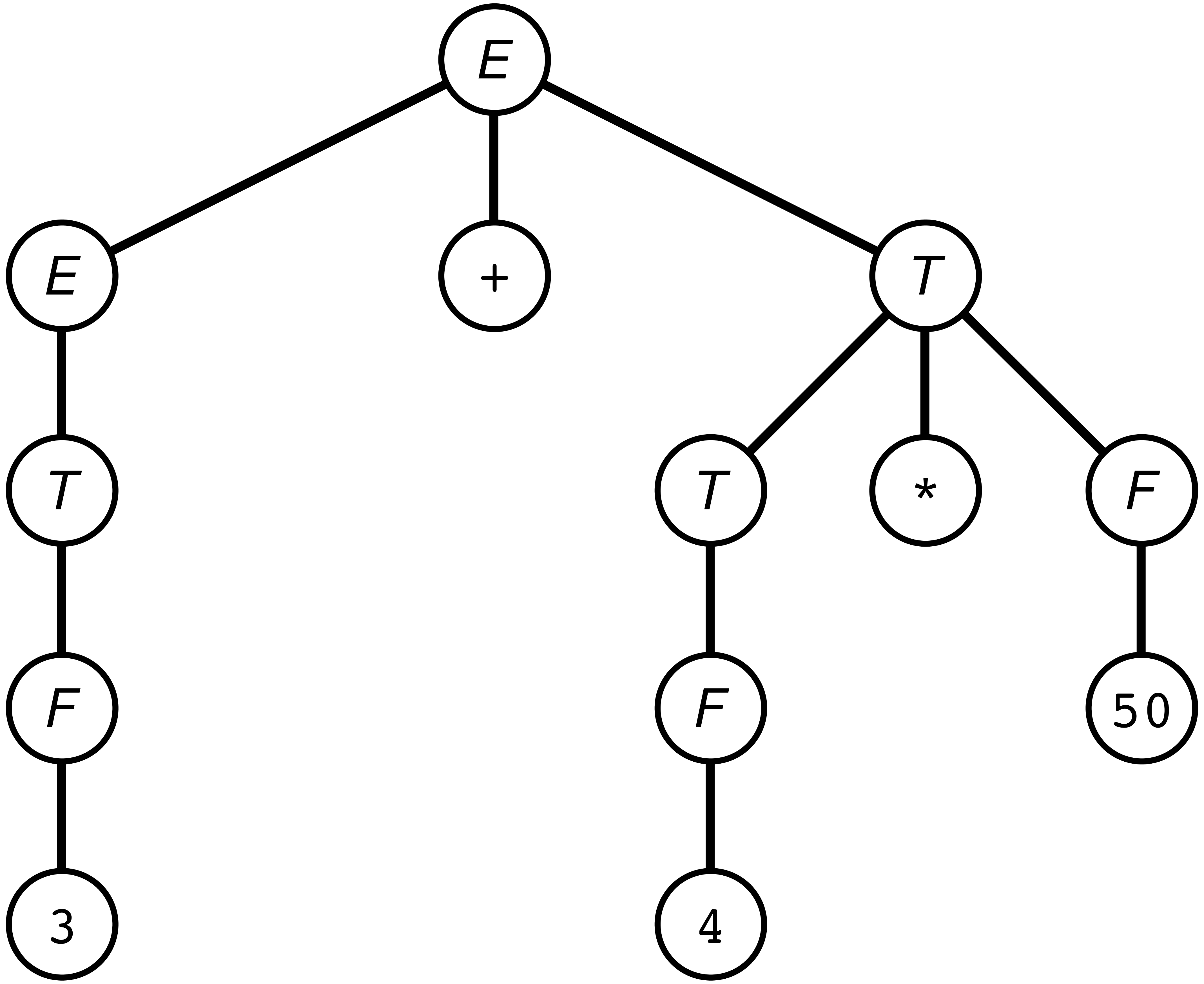
$$
\begin{aligned}
\mathit{EXP} &\Rightarrow \mathit{EXP} + \mathit{TERM} \\
&\Rightarrow \mathit{TERM} + \mathit{TERM} \\
&\Rightarrow \mathit{FACTOR} + \mathit{TERM} \\
&\Rightarrow \texttt{3} + \mathit{TERM} \\
&\Rightarrow \texttt{3} + \mathit{TERM} * \mathit{FACTOR} \\
&\Rightarrow \texttt{3} + \mathit{FACTOR} * \mathit{FACTOR} \\
&\Rightarrow \texttt{3} + \texttt{4} * \mathit{FACTOR} \\
&\Rightarrow \texttt{3 + 4 * 50}
\end{aligned}
$$

$$
\boxed{
\begin{aligned}
&\mathit{EXP} \rightarrow \mathit{EXP} + \mathit{TERM} \mid \mathit{TERM} \\
&\mathit{TERM} \rightarrow \mathit{TERM} * \mathit{FACTOR} \mid \mathit{FACTOR} \\
&\mathit{FACTOR} \rightarrow (\ \mathit{EXP}\ ) \mid \text{number}
\end{aligned}
}
$$

Parse tree

Corresponds to the left-most derivation

$$
\begin{aligned}
EXP &\Rightarrow EXP + TERM \\
&\Rightarrow TERM + TERM \\
&\Rightarrow FACTOR + TERM \\
&\Rightarrow \mathtt{3} + TERM \\
&\Rightarrow \mathtt{3} + TERM * FACTOR \\
&\Rightarrow \mathtt{3} + FACTOR * FACTOR \\
&\Rightarrow \mathtt{3} + \mathtt{4} * FACTOR \\
&\Rightarrow \mathtt{3} + \mathtt{4} * \mathtt{50}
\end{aligned}
$$

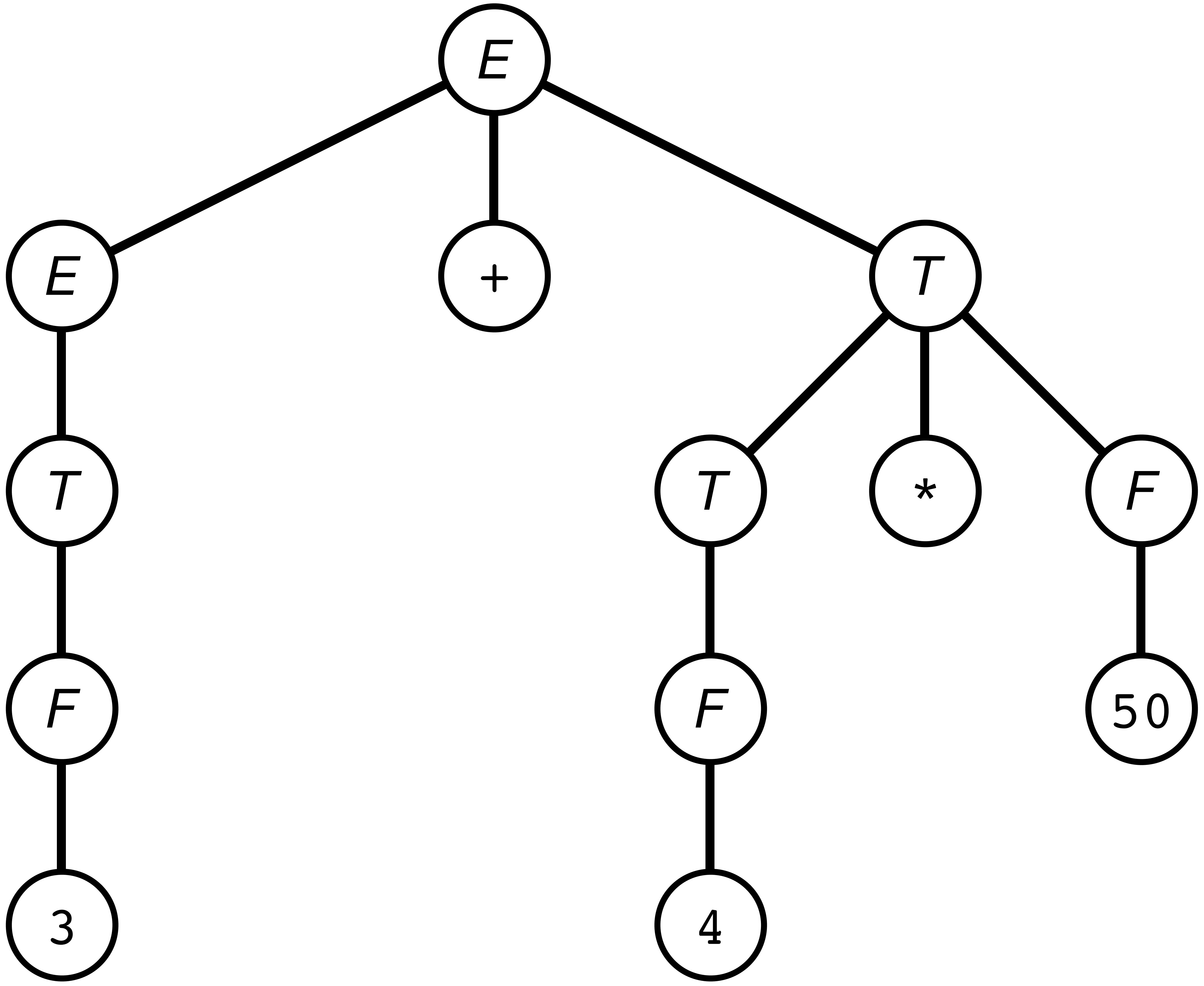


Note that the derived expression is a left-to-right traversal of the leaves

Parse tree

The structure of the tree encodes the order of operation

It's clear that we have to evaluate the 4 * 50 before we can add to the 3



The language generated by a grammar

One nonterminal is designated as the start nonterminal

- Typically, this is the nonterminal on the left-hand side of the first production rule

The language *generated* by the grammar is the set of words over the terminal alphabet which can be derived by the production rules, starting with the start nonterminal

Given our grammar for arithmetic

- `1 * ( 2 + 3 )` is in the language generated by the grammar
- `85 + * 10` is not

Why do we care (in this class)?

We're going to start with a (structured) list that represents our programs, `exp`

`(parse exp)` is going to parse that list into a tree

`(eval-exp tree environment)` will evaluate the `tree` in the `environment`

We can represent all of the syntactically valid Scheme expressions MiniScheme supports on a single slide using a grammar

A convenient shorthand

It's often useful to say that a particular terminal or nonterminal can appear 0 or more times
$A \rightarrow xA \mid \varepsilon$

where x is either a terminal or nonterminal and ε represents the empty word

Similarly, it's often useful to say that a particular terminal or nonterminal can appear 1 or more times
$A \rightarrow xA \mid x$

We write x^* or x^+ as a shorthand for these constructs

A full grammar for Minischeme

EXP → number
 | symbol
 | (`if` *EXP EXP EXP*)
 | (`let` (*LET-BINDINGS*) *EXP*)
 | (`letrec` (*LET-BINDINGS*) *EXP*)
 | (`lambda` (*PARAMS*) *EXP*)
 | (`set!` symbol *EXP*)
 | (`begin` *EXP**)
 | (*EXP*+)

LET-BINDINGS → *LET-BINDING**

LET-BINDING → [symbol *EXP*]

PARAMS → symbol*