Programming Abstractions

Lecture 17: MiniScheme Introduction and grammars

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:

```
(struct env (syms vals previous) #:transparent)
```

Environment data type

Constructor for extending an environment

```
(env list-of-syms list-of-vals previous-env)
```

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

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

The top-level environment can now be

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

Provided procedures

```
; Extended environment recognizer.
(env? e)

; Accessors
(env-syms e)
(env-vals e)
(env-previous e)
```

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`

A quick introduction to grammars

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

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

Grammars, slightly more formally

A grammar is a set of rules that describe how to *generate* a string

Grammar have three basic components

- A set of variables or **nonterminals** which expand into strings (examples soon)
- A set of **terminal symbols** from which the final word is to be constructed
- A set of **production rules** which describe how a nonterminal can be expanded

Simple example: arithmetic

EXP → *EXP* + *EXP*
EXP → *EXP* * *EXP*
EXP → (*EXP*)
EXP → *NUM*
NUM → *D NUM*
NUM → *D*
D → 0
D → 1
⋮
D → 9

Nonterminals: *EXP*, *NUM*, *D*
Terminals: +, *, (,), 0, 1, …, 9

Starting with *EXP* we can generate strings like

- 0
- 10 + 82 * 15
- (103 + (27 * 8)) * (6 + 9)

We cannot generate strings like

- 3 * + 5
- (3 + 8
- +
- *

This grammar is ambiguous

There are multiple ways to generate the string 1 * 5 + 8

$$
\begin{aligned}
EXP &\Rightarrow EXP + EXP \\
&\Rightarrow EXP * EXP + EXP \\
&\Rightarrow NUM * EXP + EXP \\
&\Rightarrow D * EXP + EXP \\
&\Rightarrow 1 * EXP + EXP \\
&\Rightarrow 1 * NUM + EXP \\
&\Rightarrow 1 * D + EXP \\
&\Rightarrow 1 * 5 + EXP \\
&\vdots \\
&\Rightarrow 1 * 5 + 8
\end{aligned}
$$

$$
\begin{aligned}
EXP &\Rightarrow EXP * EXP \\
&\Rightarrow NUM * EXP \\
&\Rightarrow D * EXP \\
&\Rightarrow 1 * EXP \\
&\Rightarrow 1 * EXP + EXP \\
&\Rightarrow 1 * NUM + EXP \\
&\Rightarrow 1 * D + EXP \\
&\Rightarrow 1 * 5 + EXP \\
&\vdots \\
&\Rightarrow 1 * 5 + 8
\end{aligned}
$$

Unambiguous grammars

Sometimes, we can design a better grammar which is not ambiguous (but not always!)

This can let us give meaning to strings

E.g., `1 * 5 + 8` *should* really mean (1 * 5) + 8 and not 1 * (5 + 8)

A better grammar for arithmetic

EXP → *EXP* + *TERM*
EXP → *TERM*
TERM → *TERM* * *FACTOR*
TERM → *FACTOR*
FACTOR → (*EXP*)
FACTOR → *NUM*
NUM → *D NUM*
D → 0
D → 1
⋮
D → 9

Compact form:

EXP → *EXP* + *TERM* | *TERM*
TERM → *TERM* * *FACTOR* | *FACTOR*
FACTOR → (*EXP*) | *NUM*
NUM → *D NUM*
D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Parse trees

Grammars give rise to parsers which can parse strings into trees

With good choices of grammars, we can derive meaning from the parse tree

Parse tree for 3 + 4 * 50

EXP ⇒ *EXP* + *TERM*

⇒ *TERM* + *TERM*

⇒ *FACTOR* + *TERM*

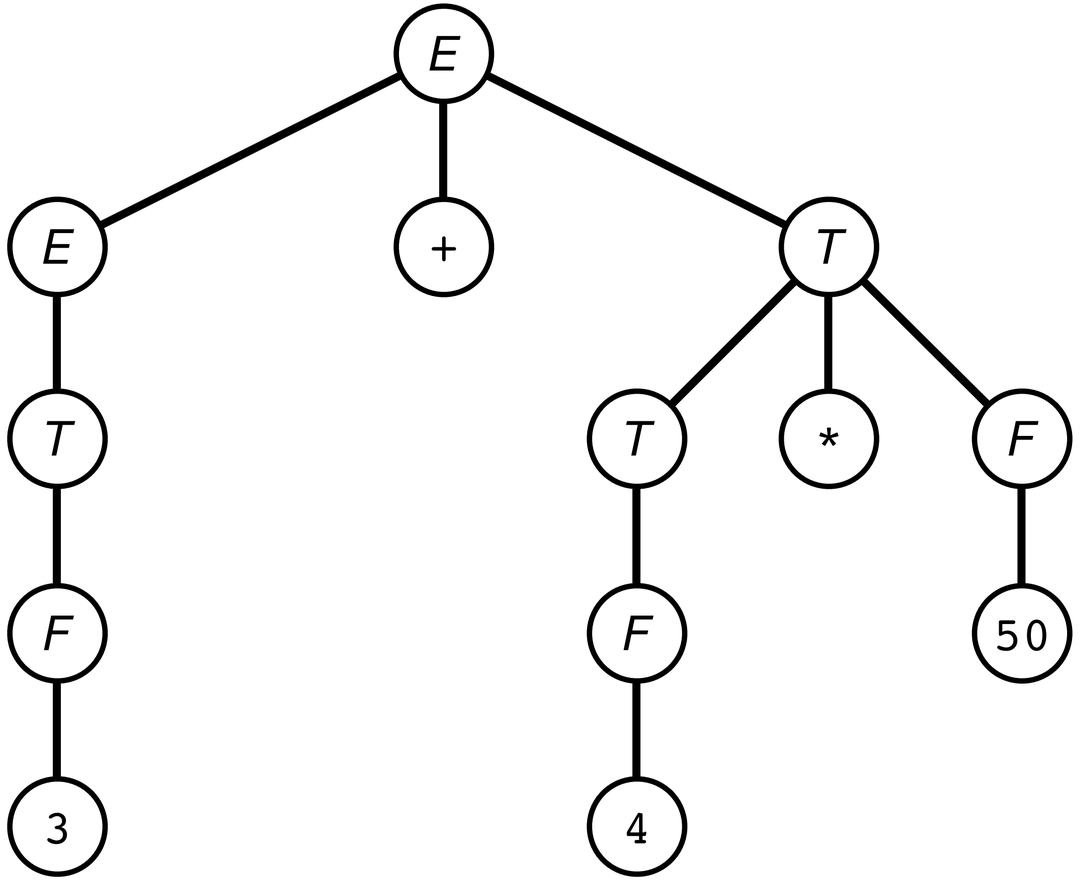
⇒ 3 + *TERM*

⇒ 3 + *TERM* * *FACTOR*

⇒ 3 + *FACTOR* * *FACTOR*

⇒ 3 + 4 * *FACTOR*

⇒ 3 + 4 * 50

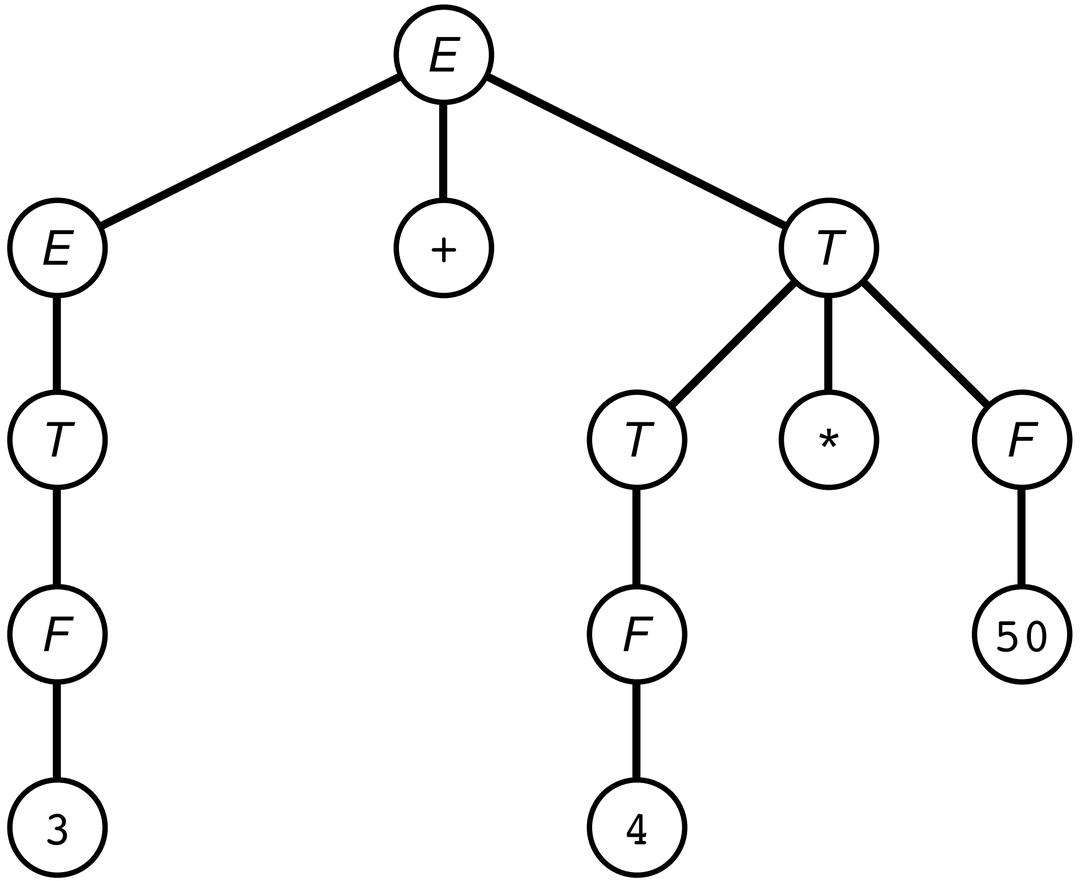


(I omitted the rules for NUM and D)

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



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

Why do we care (in 275)?

We're going to specify a grammar for MiniScheme

We'll use this to

- specify what needs to be implemented in each part
- a guide for how MiniScheme should be parsed

There's a strong connection between grammars and parsing which we won't explore in this course

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*

Can

```
(if (if 0 1 2)
    (if 3 4 5)
    (if x y z))
```

be generated by the grammar for MiniScheme?

A. Yes

B. No. (if …) cannot appear as the first expression of another if

C. No. (if …) cannot appear as the "then" or "else" expressions in another if

D. No. `x`, `y`, and `z` aren't defined

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*

Syntactically valid

Consider the invalid Scheme program

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

This is *syntactically* valid (i.e., it's a valid string generated by the MiniScheme grammar) but *semantically* meaningless as we don't have a binding for the identifier `z`