CSCI 275: Programming Abstractions

Lecture 17: MiniScheme A, B & Environments

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Questions? Concerns?

- The break counts as a weekend for any extension requests

- Start thinking about MiniScheme project teams!
  - Three homeworks with the **<u>same team</u>**

Reminder: MiniScheme Project

You're going to *build an interpreter* for a subset of Scheme (called MiniScheme)

What does an interpreter do? *Executes* a program

Grammar

We need a way to specify the language of a valid program

Parser

We need to determine if a given program is valid (a tree!)

Evaluator

We need to evaluate a given program

Literals & Symbols

Numbers first

EXP → number   parse into `lit-exp`

We're going to need a data type to represent literal expression (and the only type of literals we have are numbers)

We're going to want something which gives

```
(lit-exp num) ; constructor
(lit-exp? exp) ; recognizer
(lit-exp-num exp) ; accessor
```

Parsing Numbers

```
(define (parse input)
  (cond [(number? input) (lit-exp input)]
        [else
       (error 'parse "Invalid syntax ~s" input)]))
```

Throwing errors is **important** in MiniScheme!

MiniScheme: *You don't need to implement it exactly the way I do in class, feel free to code how you'd like but you do need to use the name parse*

We are *implementing* MiniScheme using Racket. The code about is Racket!

Think about how Python is implemented in C.

What does `(parse 15)` return, assuming the implementation we've discussed so far?

A. `15`

B. `(number 15)`

C. `(lit-exp 15)`

D. `(lit-exp "15")`

E. It's an error of some sort

Why is `(lit-exp 15)` what we want? In other words, why is there a data type for a number in our parser?

A. We just like to complicate things in this class

B. We parse *everything* into a tree, so we need a node to “hold” numbers/etc.

C. This relates to the grammar we talked about previously

D. More than one of the above

E. No idea

Evaluating literals

A starting interpreter:

```
(define (eval-exp tree e)
  (cond [(lit-exp? tree) (lit-exp-num tree)]
        [else
          (error 'eval-exp "Invalid tree: ~s"
                 tree)]))
```

What does `(eval-exp 15 empty-env)` return, assuming the implementation we've discussed so far)?

A. `15`

B. `(value 15)`

C. `(lit-exp 15)`

D. It's an error of some sort

```
(define (eval-exp tree e)
   (cond [(lit-exp? tree)
         (lit-exp-num tree)]
               [else
         (error 'eval-exp
   "Invalid tree: ~s" tree)]))
```

What does `(eval-exp (lit-exp 15) empty-env)` return, assuming the implementation we've discussed so far?

A. `15`

B. `(value 15)`

C. `(lit-exp 15)`

D. It's an error of some sort

```
(define (eval-exp tree e)
   (cond [(lit-exp? tree)
          (lit-exp-num tree)]
                [else
          (error 'eval-exp
   "Invalid tree: ~s" tree)]))
```

Putting them together again

```
> (parse 107)
(lit-exp 107)

> (lit-exp 107)
(lit-exp 107)

> (eval-exp (lit-exp 107) empty-env)
107

> (eval-exp (parse 107) empty-env)
107
```

Recall: How to implement MiniScheme

For each new type of expression:

- Add a new data type
  - `ite-exp`
  - `let-exp`
  - etc.
- Modify `parse` to produce those
- Modify `eval-exp` to interpret them

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

Remember: writing in Racket, implementing MiniScheme

Let's add some symbols (`'a`, `'+`, etc.) !

Grammar

EXP → number   parse into `lit-exp`

      | symbol   parse into `var-exp`

Data type for a variable reference expression might have:

```
(var-exp symbol) ;constructor
(var-exp? exp)   ;recognizer
(var-exp-symbol exp) ;accessor
```

Remember that numbers parse to `lit-exp` expressions.

What do we want `(parse 'x)` to return?

A. `10`

B. `(lit-exp 10)`

C. `(lit-exp 'x)`

D. `(var-exp 10)`

E. `(var-exp 'x)`

Let’s say we want to run

`(eval-exp (parse 'x) ... ).`

What makes this different than evaluating a number?

How do we know what `x` means?

We bind things frequently in Racket: we make calls to `let`, we bind parameters of `lambda`s, etc.

Big Idea: to be able to find what a variable is bound to, we need a map from variables to their bound values. This is called an *environment*!

We've discussed this a bit before!

Recall that when Racket evaluates a variable, the result is the value that the variable is bound to

If we have `(define x 10)`, then evaluating `x` gives us the value `10`

If we have `(define (foo x) (- x y))`, then evaluating `foo` gives us the procedure `(lambda (x) (- x y))`, along with a way to get the value of `y` (which is hopefully defined!)

Racket needs a way to look up values that correspond to variables: an **environment**

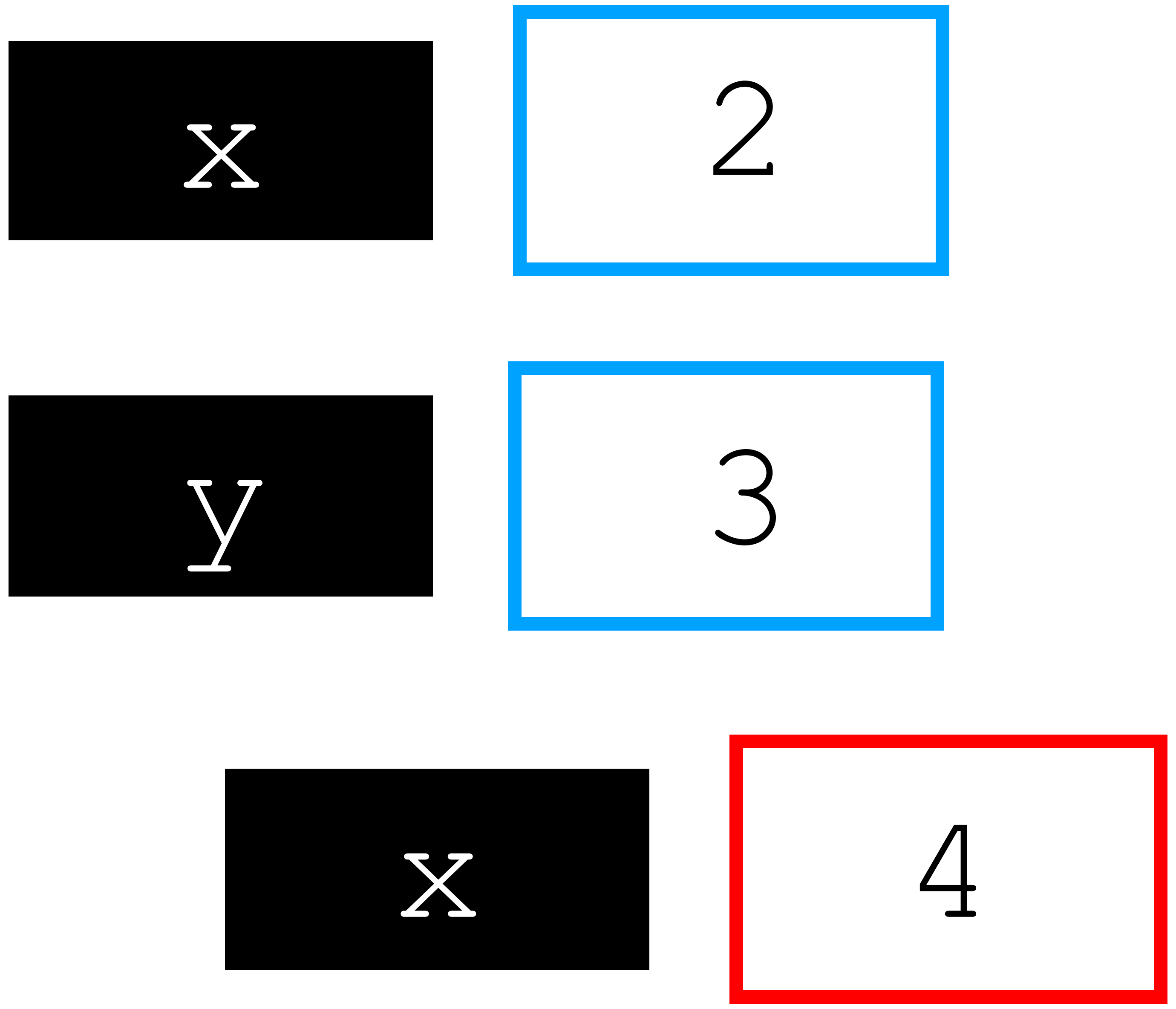
Your Task: Build an Environment!

You will build an environment (HW5) and there are rules for Racket about how variable binding works

- You have been mentally developing such mappings already as you trace through program evaluation!

Environments: Examples

```
(let ([x 2]
      [y 3])
  (let ([x 4])
    (+ x y)))
```



When we execute the following, what is the result?

```
(let ([x 2] [y 3])
    (let ([f (lambda (x) (+ x y))])
      (f 5)))
```

A. 8

B. 7

C. 5

D. Something else

Environment Operations

Two basic operations on environments, both of which you'll implement in MiniScheme:

1. **Look something up**
   - What is the binding of `x` right now?

1. **Add something to the environment**
   - Specifically, we'll do this by *extending* a previously known environment

(1) Look Up in Environments

We need to look up the value bound to a symbol:

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

should return 9 since the innermost binding of x is 4.

We say the inner `x` *shadows* the outer `x` – we need to account for this!

(2) Create New Environments

Create new environments by *extending* existing ones.

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