Programming Abstractions

Lecture 17: MiniScheme Introduction

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 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 (empty? 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")]))
```

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*