Programming Abstractions

Lecture 25: MiniScheme G

Stephen Checkoway

Announcement

Homework 7 is now up on the website

- Use the same groups as before (this time, they should be created already)

Exam 2 is next week

- Friday, Apr. 29: Exam 2 review; come prepared with questions!
- Monday, May 2: Exam 2, take home exam

Office hours

- Tomorrow at 13:30–14:30

`set!` and `begin` expressions

MiniScheme G: set! and begin

EXP → number	parse into `lit-exp`
| symbol	parse into `var-exp`
| (if *EXP* *EXP* *EXP*)	parse into `ite-exp`
| (let (*LET-BINDINGS*) *EXP*)	parse into `let-exp`
| (lambda (*PARAMS*) *EXP*)	parse into `lambda-exp`
| (set! symbol *EXP*)	parse into `set-exp`
| (begin *EXP**)	parse into `begin-exp`
| (*EXP* *EXP**)	parse into `app-exp`

LET-BINDINGS → *LET-BINDING**

LET-BINDING → [symbol *EXP*]*

PARAMS → symbol*

What is the value of

```
(let ([x 10])
  (+ x
     (let ([x 20])
       x)
     x))
```

This is the sum of 3 numbers

A. 30

B. 40

C. 50

D. 60

What is the value of

```
(let ([x 10])
  (+ x
     (begin
       (set! x 20)
       x)
     x))
```

This is the sum of 3 numbers

A. 30

B. 40

C. 50

D. 60

Assignments

Assignment expressions are different in nature than the functional parts of MiniScheme

The `set!` expression introduces mutable state into our language

We're going to use a Scheme `box` to model this state

Boxes in Scheme

`box` is a data type that holds a mutable value

- Constructor: `(box val)`
- Recognizer: `(box? obj)`
- Getter: `(unbox b)`
- Setter: `(set-box! b val)`

Example usage

We can create a `box` holding the value 275 with
`(define b (box 275))`

We can get the value in the box with `(unbox b)`

We can change the value in the box with `(set-box! b 572)`

If we use `(unbox b)` afterward, it'll return 572

This models the way variables work in non-functional languages

What does this code print out (ignoring line breaks) and why?

```
(define (f b)
  (displayln (unbox b))
  (set-box! b (* 2 (unbox b))))
(let ([x (box 5)])
  (f x)
  (f x)
  (displayln (unbox x)))
```

A. `5 5 5` because each call to f creates a new box (pass by value)

B. `10 10 5` because f doubles the value in the box b but box x contains 5

C. `5 10 5` because box b is initialized with value 5 but is doubled by the first call to f

D. `5 10 20` because b and x point to the same box whose value is doubled twice

Implementing set!

To implement set! in MiniScheme

- Change the environment so that *everything* in the environment is in a box
- When we evaluate a `var-exp`, we'll lookup the variable in the environment, `unbox` the result, and return it
- When we evaluate a set expression such as `(set! x 23)`, we'll lookup `x` in the environment to get its box and then set the value using `set-box!`

We can do this in four simple steps

Implementing `set!`

Step 1

We need to box every value in the environment

Find every place you extend the environment and replace each call

```
(env syms vals old-env)
```

with

```
(env syms (map box vals) old-env)
```

Implementing `set!`

Step 2

Do *not* change your `env-lookup` procedure

Do change the line in eval-exp that evaluates var-exp expressions to

```
[(var-exp? tree) (unbox (env-lookup e (var-exp-sym tree)))]
```

At this point, the interpreter should work exactly as it did before you introduced boxes!

Implementing `set!`

Step 3

Set expressions have the form `(set! sym exp)`

You need a new data type for these, I used `set-exp`

When parsing, put the unparsed symbol (i.e., `'x` rather than `(var-exp 'x)`) into the `set-exp` and the parsed expression `exp`

Implementing `set!`

Step 4

Inside eval-exp, you'll need some code

```
[(set-exp? tree)
 (set-box! (env-lookup …)
           (eval-exp …))]
```

We changed all calls to `env` to put the values in boxes but didn't change `env-lookup` to `unbox` the result when looking it up which forced us to add a call to `unbox` when handling `var-exps`. Could we have added the `unbox` to `env-lookup` instead? Why or why not?

A. No. Handling set! requires env-lookup to return a box it can modify

B. No. Primitive procedures and closures don't need to be boxed so unboxing them would be wrong

C. Yes. Every call to `env-lookup` will have to `unbox` so doing it in `env-lookup` simplifies the code.

D. Yes. We could; however, separation of concerns dictates that the code that's putting boxed values in the environment should also be responsible for unboxing them so unboxing in env-lookup is a bad idea.

What value should `(set! x 10)` return in MiniScheme?

A. The original value of x

B. The new value of x (10 in this case)

C. `False`

D. `null`

E. Nothing (which Racket calls `void`)

Let's make set! useful!

MiniScheme now has set! but it isn't of much use until we can execute a sequence of expressions like

```
(let ([x 0])
  (begin
    (set! x 23)
    (+ x 5)))
```

In Racket, we don't need the `begin`, but we do in MiniScheme because our let expressions only have a single expression as a body

Parsing a `begin` expression

`(begin exp1 exp2 ... expn)`

You need a new data type to hold these

- `begin-exp` is a good name

The expressions in `(begin exp1 exp2 … expn)` are evaluated in order and the value of the expression is the value that results from evaluating `expn`. How should we implement evaluating all the expressions? Assume we have something like `(let ([exps (begin-exp-exps tree)]) …)`.

A. `(map eval-exp exps)`

B. `(map (λ (exp) (eval-exp exp e)) exps)`

C. `(foldr (λ (exp acc) (eval-exp exp e)) (void) exps)`

D. `(foldl (λ (exp acc) (eval-exp exp e)) (void) exps)`

E. More than one of the above

Evaluating a `begin` expression

`(begin exp1 exp2 ... expn)`

Evaluate each expression in turn, returning the final one

- You can create a helper function to do that, or you can use our old friend: `foldl`
- My code looks something like `(foldl (λ (exp acc) (eval-exp exp e)) (void) …)`
- `(void)` returns, well, a void value which does nothing