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)
- It's due on Dec. 17

Exam 2 is next week

- Monday, Dec. 13: Exam 2 review; come prepared with questions!
- Wednesday, Dec. 15: Exam 2, take home exam

Office hours

- Tomorrow at 13:30–14:30

Example: `((lambda (x y) (+ x y)) 3 5)`

Parsing

Parse into an `(app-exp proc args)`

```
(app-exp (lambda-exp '(x y)
                     (app-exp (var-exp '+)
                              (list (var-exp 'x)
                                    (var-exp 'y))))
         (list (lit-exp 3)
               (lit-exp 5)))
```

Example: `((lambda (x y) (+ x y)) 3 5)`

Evaluating

```
(app-exp (lambda-exp '(x y)
                     (app-exp (var-exp '+)
                              (list (var-exp 'x)
                                    (var-exp 'y))))
         (list (lit-exp 3) (lit-exp 5)))
```

This is evaluated by calling apply-proc with the evaluated procedure and evaluated arguments

The procedure evaluates to

```
(closure '(x y)
         (app-exp (var-exp '+)
                  (list (var-exp 'x) (var-exp 'y)))
         e)
```

The arguments evaluate to `'(3 5)`

Example: `((lambda (x y) (+ x y)) 3 5)`

Evaluating

`apply-proc` will evaluate the closure

```
(closure '(x y)
         (app-exp (var-exp '+)
                  (list (var-exp 'x) (var-exp 'y)))
         e)
```

by calling `eval-exp` on the body in the environment `e[x ↦ 3, y ↦ 5]`

Since the body is an `app-exp`, it'll evaluate `(var-exp '+)` to get `(prim-proc '+)` and the arguments to get `'(3 5)`

Example 2

Parsing

Example 2

Parsing

What is the result of parsing this?

```
(let ([f (lambda (x) (* 2 x))])
  (f 6))
```

Example 2

Parsing

What is the result of parsing this?

```
(let ([f (lambda (x) (* 2 x))])
  (f 6))

(let-exp '(f)
         (list (lambda-exp
                 '(x)
                 (app-exp (var-exp '*)
                          (list (lit-exp 2) (var-exp 'x)))))
         (app-exp (var-exp 'f)
                  (list (lit-exp 6))))
```

Example 2

Evaluating

```
(let-exp '(f)
         (list (lambda-exp
                 '(x)
                 (app-exp (var-exp '*)
                          (list (lit-exp 2) (var-exp 'x)))))
         (app-exp (var-exp 'f)
                  (list (lit-exp 6))))
```

Evaluate the `let-exp` by extending the current environment `e` with `f` bound to the closure we get by evaluating the `lambda-exp` in environment `e`:

```
(closure '(x)
         (app-exp (var-exp '*)
                  (list (lit-exp 2) (var-exp 'x)))
          e)
```

Example 2

Evaluating

With `f` bound to

```
(closure '(x)
          (app-exp (var-exp '*)
                   (list (lit-exp 2) (var-exp 'x)))
           e)
```

we next evaluate the body of the let

```
(app-exp (var-exp 'f) (list (lit-exp 6)))
```

This will evaluate `(var-exp 'f)`, getting the closure above and evaluate the arguments getting `'(6)`

`apply-proc` will call `eval-exp` on the body of the closure and the extended environment `e[x ↦ 6]`

`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 …))]
```

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

- Since begin creates a sequence of expressions, `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