

Programming Abstractions

Week 11-1: MiniScheme F and G, lambdas and set!

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 August 18

Review: How do we parse an application like (+ 2 3)?

A. ' (app-exp + 2 3)

B. ' (app-exp + (2 3))

C. ' (app-exp (var-exp +) (lit-exp 2) (lit-exp 3))

D. ' (app-exp (var-exp +) ((lit-exp 2) (lit-exp 3)))

E. None of the above

At a higher-level of detail

Applications are parsed into two parts

- The expression for the procedure part
- The list of parsed arguments

Evaluating an app-exp

Evaluating an app-exp

How do we evaluate the app-exp we get from
`(app-exp parsed-proc list-of-parsed-args)`?

Evaluating an app-exp

How do we evaluate the app-exp we get from
(app-exp parsed-proc list-of-parsed-args)?

In steps

- ▶ We evaluate the `parsed-proc` and the `list-of-parsed-args` in the current environment
- ▶ Then we call `apply-proc` with the evaluated procedure and list of arguments

MiniScheme F: Lambdas

$EXP \rightarrow$ number
| symbol
| (if $EXP EXP EXP$)
| (let ($LET-BINDINGS$) EXP)
| (lambda ($PARAMS$) EXP)
| ($EXP EXP^*$)

$LET-BINDINGS \rightarrow LET-BINDING^*$

$LET-BINDING \rightarrow [\text{symbol } EXP]^*$

$PARAMS \rightarrow \text{symbol}^*$

parse into `lit-exp`

parse into `var-exp`

parse into `ite-exp`

parse into `let-exp`

parse into `lambda-exp`

parse into `app-exp`

Implementing lambdas

Parsing

Parse a lambda expression such as `(lambda (x y z) body)` into a new `lambda-exp` structure

This needs

- ▶ The parameter list, e.g., `(x y z)`
- ▶ the parsed body

Note that the parameter list is not parsed, it's just a list of symbols

Implementing lambdas

Evaluating

What should a lambda-exp evaluate to?

In other words, what is the result of evaluating something like
`(lambda (x) (+ x y))`?

Closures!

We need a closure data type

- `(closure parameter-list body environment)`
- `(closure? obj)`
- `(closure-params c)`
- `(closure-body c)`
- `(closure-env c)`

The `parameter-list` and the `body` come from the `lambda-exp`

The `environment` is the current environment argument to `eval-exp`

Where should the new closure data type be defined? Why?

A. `parse.rkt`

B. `interp.rkt`

C. In the same file as `prim-proc`

D. A and C

E. B and C

To recapitulate

To parse a lambda

- Make a new `lambda-exp` object to hold parameters and body

To evaluate a lambda

- Make a new `closure` object to hold the parameters, body, and environment

Nothing new is needed for parsing **calls** to lambda expressions; why?

```
(let ([f (lambda (x) (+ x y))])  
  (f (- a b)))
```

Evaluating calls to closures

Recall: All applications are evaluated by calling `apply-proc` with the evaluated procedure and the list of evaluated arguments

Here's what our `apply-proc` looks like after homework 6

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
         (apply-primitive-op (prim-proc-op proc) args)]
        [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

Evaluating calls to closures

We need to add some code before the `else`

```
(define (apply-proc proc args)
  (cond [(prim-proc? proc)
        (apply-primitive-op (prim-proc-op proc) args)]
        [(closure? proc) ...]
        [else (error 'apply-proc "bad procedure: ~s" proc)]))
```

How do we evaluate the closure?

How do we evaluate the closure?

At a high level (don't think about MiniScheme here), given a closure and some arguments, how do we evaluate calling the closure?

How do we evaluate the closure?

At a high level (don't think about MiniScheme here), given a closure and some arguments, how do we evaluate calling the closure?

Steps

- ▶ Extend the closure's environment with bindings from the closure's parameters to argument values
- ▶ Evaluate the body of the closure in this extended environment

How do we evaluate the closure?

At a high level (don't think about MiniScheme here), given a closure and some arguments, how do we evaluate calling the closure?

Steps

- ▶ Extend the closure's environment with bindings from the closure's parameters to argument values
- ▶ Evaluate the body of the closure in this extended environment

If you find yourself wanting to pass the environment from `eval-exp` to `apply-proc`, there is something wrong; you don't need to do that

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 +)
                                ((var-exp x)
                                 (var-exp y))))
  ((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 +)
                                ((var-exp x) (var-exp y))))
  ((lit-exp 3) (lit-exp 5)))
```

This is evaluated in the current environment `e` by calling `apply-proc` with the evaluated procedure and evaluated arguments

The **procedure** evaluates to

```
'(closure (x y)
  (app-exp (var-exp +)
            ((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 +)
                   ((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)  
  ((lambda-exp (x)  
            (app-exp (var-exp *)  
                      ((lit-exp 2) (var-exp x))))))  
  (app-exp (var-exp f)  
            ((lit-exp 6))))
```

Example 2

Evaluating

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

Example 2

Evaluating

With f bound to

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

we next evaluate the body of the let

```
'(app-exp (var-exp f) ((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]`

This is another application expression and the process continues

set ! expressions

MiniScheme G: set! and begin

$EXP \rightarrow$ number
| symbol
| (if EXP EXP EXP)
| (let ($LET-BINDINGS$) EXP)
| (lambda ($PARAMS$) EXP)
| (set! symbol EXP)
| (begin EXP^*)
| (EXP EXP^*)

$LET-BINDINGS \rightarrow LET-BINDING^*$

$LET-BINDING \rightarrow [\text{symbol } EXP]^*$

$PARAMS \rightarrow \text{symbol}^*$

parse into lit-exp
parse into var-exp
parse into ite-exp
parse into let-exp
parse into lambda-exp
parse into set-exp
parse into begin-exp
parse into app-exp

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

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

Two ways to do this (and I'm quoting Bob here)

- ▶ If you are young and cocky and sure you can find every place you extend the environment, you can replace each call

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

with

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

- ▶ If you have 68 years of experience with screwing up [I'm still quoting Bob here], you might prefer to change the definition of `env` to do

```
(list '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

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, I called mine `seq-exp` but `begin-exp` is also a good name (and visually distinct from `set-exp`)

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