# Programming Abstractions

## Week 11-2: MiniScheme G and H, set! and letrec

**Stephen Checkoway**

# 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's the value of the let expression

```
(define (double! b)
  (set-box! b (* 2 (unbox b))))

(let ([foo (box 3)])
  (double! foo)
  (double! foo)
  (double! (box 2))
  (unbox foo))
```

A. 3

B. 4

C. 6

D. 12

E. 24

# 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 sims (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

# MiniScheme H: Recursion

Review: What is the value of this expression?

```
(let ([f add1])
  (let ([f (λ (x)
             (if (= x 0)
                 10
                 (* 2 (f 0))))])
    (f 3)))
```

A. 2

B. 4

C. 10

D. 20

E. An error

What is the result of this expression?

```
(let ([f (λ (n)
           (if (= 0 n)
               empty
               (cons n (f (- n 1)))))])
  (f 4))
```

A. `'(0 1 2 3 4)`

B. `'(1 2 3 4)`

C. `'(4 3 2 1 0)`

D. `'(4 3 2 1)`

E. An error

# Implementing recursion in MiniScheme H

`(letrec ([f exp1] [g exp2] ...) body)`

We'll have the parser parse a letrec expression into something equivalent that uses only things we have implemented

We won't need to change `eval-exp` at all!

# Two options

We can use the Y combinator (technically the Z combinator)

We can use `set!/begin`

Which would you prefer?

# Z combinator it is!

```
Z = λf.(λx.f(λ v.xxv))(λx.f(λv.xxv))
```

Translated from λ-calculus to Scheme, we have

# Just kidding, let's use set!/begin

What does this evaluate to?

```
(let ([f 0])
  (let ([g 34])
    (begin
     (set! f g)
     f)))
```

# How about this?

What does this evaluate to?

```
(let ([f 0])
  (let ([g (λ (x) (+ 1 x))])
    (begin
     (set! f g)
     (f 5))))
```

# And this?

What does this evaluate to?

```
(let ([f 0])
  (let ([g (λ (x) (if (< 9 x) 10 (f (add1 x))))])
    (begin
     (set! f g)
     (f 5))))
```

# Write factorial without letrec

```
(let ([fact 0])
  (let ([placeholder (λ (n)
                         (if (= n 0)
                             1
                             (* n (fact (sub1 n)))))])
    (begin
     (set! fact placeholder)
     (fact 5))))
```

# Mutual recursion

```
(letrec ([even? (lambda (x)
                  (cond [(= 0 x) #t]
                        [(= 1 x) #f]
                        [else (odd? (- x 1))]))]
         [odd? (lambda (x)
                 (cond [(= 0 x) #f]
                       [(= 1 x) #t]
                       [else (even? (- x 1))]))])
  (odd? 23))
```

# Mutual recursion without `letrec`

```
(let ([even? 0]
      [odd? 0])
  (let ([f (lambda (x)
             (cond [(= 0 x) #t]
                   [(= 1 x) #f]
                   [else (odd? (- x 1))]))]
        [g (lambda (x)
             (cond [(= 0 x) #f]
                   [(= 1 x) #t]
                   [else (even? (- x 1))]))])
    (begin
     (set! even? f)
     (set! odd? g)
     (odd? 23))))
```

# General transformation

Replace

```
(letrec ([f1 exp1] … [fn expn])
  body)
```

with

```
(let ([f1 0] … [fn 0])
  (let ([g1 exp1] … [gn expn])
    (begin
     (set! f1 g1)
     …
     (set! fn gn)
     body)))
```

# General transformation

Replace

```
(letrec ([f1 exp1] … [fn expn])
  body)
```

with

```
(let ([f1 0] … [fn 0])
  (let ([g1 exp1] … [gn expn])
    (begin
     (set! f1 g1)
     …
     (set! fn gn)
     body)))
```

We need some new symbols!

# Generating symbols

`(gensym)`

We can use `(gensym)` to generate new, unused symbols

```
> (gensym)
'g75075
> (gensym)
'g75106
```

# Final MiniScheme grammar

| | | |
|---|---|---|
| *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` |
| \| ( letrec ( *LET-BINDINGS* ) *EXP* ) | transform into equivalent `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*

# Parsing letrec expressions

`(letrec ([f1 exp1] … [fn expn]) body)`

We have three parts

- `syms = (f1 … fn) = (map first (second input))`
- `exps = (exp1 … expn) = (map second (second input))`
- `body = (third input)`

We need to construct several parts from these

- The outer let: `(let ([f1 0] … [fn 0]) …)`
- The inner let: `(let ([g1 exp1] … [gn expn]) …)`
- The set!s: `(begin (set! f1 g1) … (set! fn gn) …)`

# The outer let

`(let ([f1 0] … [fn 0]) …)`

Recall that our let-exp has a list of symbols, a list of parsed expressions, and a parsed body

We already got the symbols: `(f1 … fn) = syms`

For the parsed expressions: `(map (λ (s) (lit-exp 0)) syms)`

The parsed body is going to be another `let-exp`

# The inner let

`(let ([g1 exp1] … [gn expn]) …)`

For the symbols: `new-syms = (map (λ (s) (gensym)) syms)`

For the parsed expressions: `(map parse exps)`

The parsed body is a begin expression

# The begin expression

`(begin (set! f1 g1) … (set! fn gn) body)`

Recall that `begin-exp` takes a list of parsed expressions

Three reasonable options

- Generate the `set!`s via `(map (λ (s new-s) …) syms new-syms)`
  Append `(list (parse body))`
- Write your own recursive procedure to build the list
- Use `foldr`

```
(foldr (λ (s new-s acc)
         (cons … acc))
       (list (parse body))
       syms
       new-syms)
```

# A (mostly) complete example

```
(letrec ([length (lambda (lst)
                   (if (null? lst)
                       0
                       (add1 (length (cdr lst)))))])
  (length (list 10 20 30)))
```

parses to

```
'(let-exp (length)
          ((lit-exp 0))
          (let-exp (g75784)
                   ((lambda-exp (lst) (ite-exp …)))
                   (begin-exp
                    ((set-exp length (var-exp g75784))
                     (app-exp (var-exp length) (…))))))
```

# And that's it!

We don't need to change eval-exp at all because we already know how to evaluate let-, set-, and begin-expressions.