

# **Programming Abstractions**

**Week 11-2: MiniScheme H**

**Stephen Checkoway**

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!

$$z = \lambda f. (\lambda x. f (\lambda v. xxv)) (\lambda x. f (\lambda v. xxv))$$

Translated from  $\lambda$ -calculus to Scheme, we have

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

To what does this evaluate?

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

A. 0

B. 34

C. An error

To what does this evaluate?

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

A. 0

B. 1

C. 5

D. 6

E. An error

To what does this evaluate?

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

A. 0

B. 5

C. 10

D. 11

E. An error



# Let's draw the environments

```
(let ([f 0])
  (let ([g (λ (x)
             (if (>= x 10)
                 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? (sub1 x))])))
         [odd? (lambda (x)
                (cond [(= 0 x) #f]
                      [(= 1 x) #t]
                      [else (even? (sub1 x))])))])
  (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
```

# A common mistake with gensym

Every time you call (gensym), you get a new symbol

If you transform (letrec ([f ...]) ...) into

```
(let ([f 0])  
  (let ([(gensym) ...])  
    (begin  
      (set! f (gensym))  
      ...)))
```

your code will fail to work because the two symbols will be different!



# Final MiniScheme grammar

$EXP \rightarrow$	number	parse into <code>lit-exp</code>
	symbol	parse into <code>var-exp</code>
	( if $EXP EXP EXP$ )	parse into <code>ite-exp</code>
	( let ( $LET-BINDINGS$ ) $EXP$ )	parse into <code>let-exp</code>
	( letrec ( $LET-BINDINGS$ ) $EXP$ )	transform into equivalent <code>let-exp</code>
	( lambda ( $PARAMS$ ) $EXP$ )	parse into <code>lambda-exp</code>
	( set! symbol $EXP$ )	parse into <code>set-exp</code>
	( begin $EXP^*$ )	parse into <code>begin-exp</code>
	( $EXP EXP^*$ )	parse into <code>app-exp</code>
$LET-BINDINGS \rightarrow$	$LET-BINDING^*$	
$LET-BINDING \rightarrow$	[ symbol $EXP$ ] <sup>*</sup>	
$PARAMS \rightarrow$	symbol <sup>*</sup>	

# 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`.