CSCI 275: Programming Abstractions

Lecture 26: MiniScheme H (`letrec`)

Spring 2025

Stephen Checkoway
Slides from Molly Q Feldman

MiniScheme G Wrap Up

What is `minischeme.rkt` for?

- A reminder that we are building the code to support a real-eval-print-loop (or REPL)
- `minischeme.rkt` uses your `parse` and `eval-exp` to give you the experience of writing an expression in MiniScheme and seeing it evaluate

```
(println (eval-exp init-env (parse input)))
```

```
Welcome to DrRacket, version 8.5 [cs].
Language: racket, with debugging; memory limit: 512 MB.
MS> (let ([x 3]) (+ x 4))
7
```

Let's make `set!` useful: introduce `begin`

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

You will need a field that holds the list of parsed expressions

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 (lambda (exp) (eval-exp exp e)) exps)`

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

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

E. More than one of the above

MiniScheme H – The End!

MiniScheme H

- Go over how to implement `letrec` using nested `let`s, `set!`, and `begin`
- With that, MiniScheme key ideas are done and we've covered all the concepts for Homework 8!

What is the value of this expression in Racket?

```
(let ([f add1])
  (let ([f (lambda (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 in Racket?

```
(let ([f (lambda (n)
           (if (equal? 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

How to implement 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!

To do this, we'll use `set!/begin`

To what does this evaluate?

```
(let ([x 0])
  (let ([y 34])
    (begin
     (set! x y)
     x)))
```

A. 0

B. 34

C. An error

To what does this evaluate?

```
(let ([m 0])
  (let ([n (lambda (x) (sub1 x))])
    (begin
     (set! m n)
     (m 7))))
```

A. 0

B. -1

C. 7

D. 6

E. An error

To what does this evaluate?

```
(let ([f 0])
  (let ([g (lambda (x) (f x))])
      (begin
        (set! f add1)
        (g 3))))
```

A. 0

B. 4

C. 3

D. It runs forever

E. An error

Write factorial without letrec

```
(let ([fact 0])
  (let ([placeholder (lambda (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))))
```

How we will make this happen!

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)))
```

One problem with our plan: g1, …, gn

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)))
```

Symbols f1, …, fn are provided in the letrec
Where can we get symbols for g1, …, gn that do not conflict with existing symbols?

Generating symbols

We can use the built-in Racket command `(gensym)` to generate new, unique 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))
     ...)))
```

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

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*)	
| (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 (lambda (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 (lambda (s) (gensym)) syms)`

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

The parsed body is a `begin` expression

The begin expression

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

Three reasonable options:

1. Generate the `set-exp`s via

   `(map (lambda (s new-s) …) syms new-syms)`

   Append `(list (parse body))`
2. Write your own recursive procedure to build the list
3. Use `foldr` with *three* arguments to the `lambda`

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

Why foldr and not foldl?

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

Testing `letrec`

Problem: `(gensym)` always returns a new symbol so we can't test for equality

Solution: Test the ***structure*** of the result of `parse` is what you expect:

- Parsing a `letrec` should return a `let-exp`
- That `let-exp` should have a `let-exp` as the body
- The inner `let-exp` should have a `begin-exp` as the body
- And so on

You'll probably want to use `let-exp?`, `begin-exp?`, `set-exp?`, etc

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.