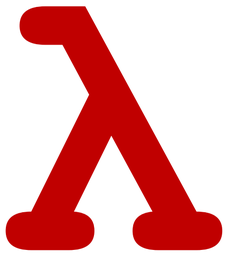
CSCI 275: Programming Abstractions

Lecture 21: MiniScheme E
Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman



Functional Language of the Week: Clojure

- Interesting combination of features: Lisp, but for the JVM and with concurrency support
- Smaller user base and support than other languages we've discussed
  - Operates in the "Benevolent Dictator for Life" Model
  - Currently supported by Nubank, the largest fintech bank in Latin America
- History of Programming Languages (HOPL) paper/talk about Clojure recently
  - HOPL is very neat if you're interested in PL & history: https://dl.acm.org/toc/pacmpl/2020/4/HOPL

Functional Language of the Week: Clojure

Racket

```
> (define (make-adder x)
    (let ([y x])
      (lambda (z) (+ y z))))
> (define add2 (make-adder 2))
> (add2 4)
6
```

Clojure

```
(defn make-adder [x]
  (let [y x]
    (fn [z] (+ y z))))
(def add2 (make-adder 2))
(add2 4)
-> 6
```

Testing

You'll need to test your implementation

Make sure you test as you go!

One test file for each MiniScheme module

- `env-tests.rkt`
- `parse-tests.rkt`
- `interp-tests.rkt`

Useful testing functions

```
; Test that expression is #t
(test-true “test name” expression)

; Test that expression is #f
(test-false “test name” expression)

; Test that (equal? actual expected) is #t
(test-equal? “test name” actual expected)
```

Assume (foo x) is a function that returns a list. Which of the following tests is the best way to test that (foo ‘z) does not contain the value ‘z?

A.
```
(test-true "foo test"
           (not (member 'z (foo 'z))))
```

B.
```
(test-false "foo test"
            (member 'z (foo 'z)))
```

C.
```
(test-equal? "foo test"
             (member 'z (foo 'z))
             #f)
```

Additional testing functions

```
; Test that (predicate expression) is #t
(test-pred “test name” predicate expression)
  — Example: (test-pred list? (foo ‘z))

; Test that expression raises an exception
(test-exn “test name”
          exn:fail?
           (lambda () expression))
```

How would we best test that (parse ‘()) raises an exception? What do the other tests do?

A.
```
(test-pred "parse empty list"
           exn:fail?
           (parse '()))
```

B.
```
(test-exn "parse empty list"
          exn:fail?
          (parse '()))
```

C.
```
(test-exn "parse empty list"
          exn:fail?
          (λ () (parse '())))
```

Parser tests

Test that you can parse numbers, symbols, applications, and if-then-else expressions

```
; Test that (var-exp? (parse 'x)) returns #t
(test-pred "Variable"
           var-exp?
           (parse 'x))

; Test that (parse 'y) returns (var-exp 'y)
(test-equal? "Variable equality"
             (parse 'y)
             (var-exp 'y))
```

Parser tests

```
; Test that (parse '()) raises exception
(test-exn "Invalid syntax ()"
          exn:fail?
          (lambda () (parse '())))

; Test that (parse "string") raises exception
(test-exn "Invalid syntax \"string\""
          exn:fail?
          (lambda () (parse "string")))
```

Interpreter tests

```
; Construct a test environment
(define test-env
  (env '(foo bar) '(10 23) init-env))

; Test evaluating literals
(test-equal? "Literal"
             (eval-exp (lit-exp 5) test-env)
             5)

; Test evaluating variables
(test-equal? "Variable"
             (eval-exp (var-exp 'foo) test-env)
             10)
```

Interpreter tests

```
; Test primitive procedures
(test-equal? "Primitive cons"
             (eval-exp (var-exp 'cons) test-env)
             (prim-proc 'cons))
```

BE CAREFUL!

Some notes on testing for Homework 6 are below.

First, don't write tests like this below if you can help it:

```
(test-equal? "Apply (- 23 3)"
             (eval-exp (parse '(- 23 3)) test-env)
             20)
```

Rather:

- Test `parse` by giving it a structured list (i.e. a MiniScheme expression)
- Test `eval-exp` by giving it a parse tree you write yourself

Why? Easy to make mistakes, sometimes unexpected behavior

An example is this (good) test for `eval-exp` for MiniScheme C:

```
(test-equal? "Apply (- 23 3)"
             (eval-exp (app-exp (var-exp '-)
                                (list (lit-exp 23)
                                      (lit-exp 3)))
                       test-env)
             20)
```

MiniScheme E: let expressions

MiniScheme E Grammar

EXP → number
| symbol
| `(` `if` *EXP EXP EXP* `)`
| `(` `let` `(` *LET-BINDINGS* `)` *EXP* `)`
| `(` *EXP EXP** `)`

LET-BINDINGS → *LET-BINDING**

LET-BINDING → `[` symbol *EXP* `]`

Parsing let expressions

```
(let ([x (+ 3 4)] [y 5] [z (foo 8)])
  body)
```

The binding list is `(second input)` where `input` is the whole let expression

The symbols are `(map first binding-list)`

These are *not* parsed – why?

The binding expressions are `(map second binding-list)`

How can we parse each of these expressions?

The body is simply `(third input)` which we can `parse`

As usual, assume we make a `let-exp` data type to hold a parsed `let` expression. What should this code return?

```
(parse '(let ([x 10]
              [y z]) y))
```

A.
```
(let-exp '(x y)
         (list (lit-exp 10) (var-exp 'z))
         (var-exp 'y))
```

B.
```
(let-exp (list (var-exp 'x) (var-exp 'y))
         (list (lit-exp 10) (var-exp 'z))
         (var-exp 'y))
```

C.
```
(let-exp (list (var-exp 'x) (var-exp 'y))
         '(10 z)
         (var-exp 'y))
```

D.
```
(let-exp '(x y) '(10 z) (var-exp 'y))
```

Example for evaluating let expressions

Consider

```
(let ([x (+ 3 4)]
      [y 5]
      [z (foo 8)])
  body)
```

To evaluate this, we need to extend the current environment with bindings for `x`, `y`, and `z` and then evaluate `body` in the extended environment

Evaluating let expressions

1. Evaluate each of the binding expressions in the `let-exp`

```
(map (lambda (exp)
       (eval-exp exp current-env))
     (let-exp-exps tree))
```

2. Bind the symbols to these values by extending the current environment

3. Evaluate the body of the let expression using the extended environment

We discussed how to extend environments before break!

How should we test the evaluation of this let expression?

```
(let ([x 10] [y 20]) y))
```

A.
```
(test-equal? "let test"
             (let ([x 10] [y 20]) y)
             20)
```

B.
```
(test-equal? "let test"
             (eval-exp (let-exp '(x y)
                                (list (lit-exp 10)
                                      (lit-exp 20))
                                (var-exp 'y))
                       test-env)
             20)
```

C.
```
(test-equal? "let test"
             (eval-exp (parse '(let ([x 10] [y 20]) y))
                       test-env)
             20)
```