Intro to Scheme

Geoffrey Matthews

January 17, 2023

Scheme

Remember to work through the tutorials!

- `https://racket-lang.org/`
  - Our implementation
- `https://docs.racket-lang.org/index.html`
  - Work through the *Quick* tutorial
- Also do one or more of the following:
  - `https://ds26gte.github.io/tyscheme/`
    - Work through Chapters 1 to 6: Recursion
    - Note that we use Racket instead of mzscheme
  - `https://www.scheme.com/tspl4/`
    - Work through Chapters 1 and 2
    - Note that we use Racket instead of chez scheme
  - `https://htdp.org/`
    - Very careful exposition.

Basic Scheme Syntax: Literal expressions

```
> 123
123
> 1.2e5
120000.0
> 1/3
1/3
> 'a
'a
> 'very-long-identifier
'very-long-identifier
> '(list of things 1 2 3)
'(list of things 1 2 3)
```

```
> "a string"
"a string"
> ")) a bad string"
")) a bad string"
> '99
99
> 99
99
> #t
#t
> '#t
#t
```

Basic Scheme Syntax: Procedure calls

```
> (+ 3 4)
7
> (+ 3 (* 4 5))
23
> (* (+ 3 4) 5)
35
> ((if #f + *) 3 4)
12
>
```

```
3 + 4 * 5
```

Basic Scheme Syntax: Naming things

```
> (define x 2)
> (define y 3)
> x
2
> y
3
> (* x y)
6
```

Basic Scheme Syntax: Conditionals

```
> (define x 2)
> (define y 3)
> (if (< x y) x y)
2
> (cond ((< x y) 'yes)
        ((> x y) 'no)
        ((= x y) 'maybe)
        (else (error "Help!" (list x y))))
'yes
>
```

Basic Scheme Syntax: Case

```
> (case (* 2 3)
    ((2 3 5 7) 'prime)
    ((1 4 6 8 9) 'composite))
'composite
>
```

Printing: three ways

```
(define astring
  "foobuz
bassssssssssssss @#$@#@$")
(display astring)
(newline)
(print astring)
(newline) ;; Notice neither prints a newline by itself
(printf
  "Formatted output: ~a and: ~a\n"
  'aThing 1234)
```

```
foobuz
bassssssssssssss @#$@#@$
"foobuz\nbassssssssssssss @#$@#@$"
Formatted output: aThing and: 1234
```

Basic Scheme Syntax: Sequencing

```
> (begin (display "4 plus 1 equals ")
         (display (+ 4 1))
         (newline))
4 plus 1 equals 5
>
```

Two ways to define procedures

```
(define (square1 x) (* x x))
(define square2 (lambda (x) (* x x)))
```

- I like to use the second way.
- The textbook uses the first way.
- Simple unit tests also supported in the `plai` language:

```
> (define testnumber 23423)
> (test (square1 testnumber)
        (square2 testnumber))
(good (square1 testnumber) 548636929 548636929 "at
    line 31")
```

Rational numbers and many other builtin types

```
(test (/ 2 3)
      2/3)
```

```
(good (/ 2 3) 2/3 2/3 "at line 35")
```

Two ways to quote things

```
(test
 '(1 2 3 a b c)
 (quote (1 2 3 a b c)))
```

```
(good '(1 2 3 a b c) '(1 2 3 a b c) '(1 2 3 a b c) "at
    line 39")
```

- We will use the first way.
- Quoted expressions denote linked lists.

Dot notation and List notation

- Make sure you can draw box and arrow diagrams for random collections of cons cells, lists, dotted lists, etc.
- Check out `boxarrow.rkt` from my github repository.

```
(test '(1 . (2 . (3 . ()))))
      '(1 2 3) )
(test (cons 3 (cons 2 (cons 1 '())))
      (list 3 2 1) )
(cons (cons 'a 'b) (list (list 'c (cons 'd 'e)
                               (cons 'f (list 'g)))))
(list 1 (list 2 (list 3 (list 4 5))))
(cons 1 (cons 2 (cons 3 (cons 4 5))))
```

```
(good '(1 2 3) '(1 2 3) '(1 2 3) "at line 49")
(good (cons 3 (cons 2 (cons 1 '()))) '(3 2 1) '(3 2 1)
    "at line 52")
'((a . b) (c (d . e) (f g)))
'(1 (2 (3 (4 5))))
'(1 2 3 4 . 5)
```

car's and cdr's can be smashed together

```
(define testlist '((a b) ((c d) e (f (g)))))
(test (cdr (car (cdr testlist)))
      (cdadr testlist))
```

`let` introduces local variables

```
(let ((x 3)
      (y 4))
  (test (* x y)
        (* 3 4)))

;; This doesn't work:

;;(let ((x 3)
;;      (y (* 5 x)))
;;   (* x y))
```

Use nested let's if you want new variables to depend on each other

```
(let ((x 3))
  (let ((y (* 5 x)))
    (* x y)))
```

Shadowed variables

```
(let ((b 99999))
  (let ((a 100)
        (b 1000)
        (c 10000))
    (+ a b c)))

(let ((b 99999))
  (+ (let ((a 100)
           (b 1000)
           (c 10000))
       (+ a b c))
     b))
```

Functions are first class values

```
(let ((f (lambda (g x) (+ (* 2 x) (g x)))))
  (f (lambda (y) (* 3 y)) 5))
```

Functions are first class values

```
(let ((f (lambda (g x) (+ (* 2 x) (g x)))))
  (f (lambda (y) (* 3 y)) 5))
```

```
(test
 (let ((+ *)) (+ 3 3))
 (* 3 3))
```

lambda and let are similar

```
(test
 (let ((x 3) (y 4)) (list 5 x y))
 ((lambda (x y) (list 5 x y)) 3 4))
```

Free variables can be captured by lambda

```
(let ((x 'sam))
  (let ((f (lambda (y z) (list x y z))))
    (f 'i 'am)))
```

- Free variables remain captured even when shadowed:

```
(let ((x 'sam))
  (let ((f (lambda (y z) (list x y z))))
    (let ((x 'mary))
      (f 'i 'am))))
```

`lambda` creates a **closure**

```scheme
(define f1
  (let ((x 'sam))
    (lambda (y z) (list x y z))))

(f1 'i 'am)
(define x 'nobody)
(f1 'i 'am)
```

```scheme
'(sam i am)
'(sam i am)
```

We can make objects with local state

```
(define counter
  (let ((n 0))
    (lambda ()
      (set! n (+ n 1))
      n)))

(counter)
(counter)
(counter)
(counter)
```

```
1
2
3
4
```

Functions can return functions

```
(define adder
  (lambda (x)
    (lambda (y) (+ x y))))

(define f (adder 10))
(define g (adder 100))
(list (f 3) (f 5) (g 3) (g 5))
```

```
'(13 15 103 105)
```

- This particular example is called **Currying** a function–taking a many-argument function and turning it into a one argument function.

We can use closures to store local data

```
(define triple
  (lambda (a b c)
    (lambda (op)
      (cond ((eqv? op 'first) a)
            ((eqv? op 'second) b)
            ((eqv? op 'third) c)))))

(define a (triple 5 9 20))
(define b (triple 'hello 'goodbye 'whatever))
(list (a 'first) (b 'second) (a 'third) (b 'first))
```

```
'(5 goodbye 20 hello)
```

We can use closures to create objects like stacks

```
(define stack
  (lambda ()
    (let ((the-stack '()))
      (lambda (op . args)
        (cond ((eq? op 'push)
               (set! the-stack (cons (car args)
                                     the-stack)))
              ((eq? op 'pop)
               (let ((top (car the-stack)))
                 (set! the-stack (cdr the-stack))
                 top))
              (else
               (error "Unknown stack operator: ~a"
                      op)))))))
(define s (stack))
(s 'push 99)
(s 'push 101)
(list (s 'pop) (s 'pop))
```

```
'(101 99)
```

List recursion

```
(require racket/trace)
(define list-length
  (lambda (lst)
    (if (null? lst) 0
        (+ 1 (list-length (cdr lst))))))
(trace list-length)
(list-length '(a b c d))
```

```
(list-length '(a b c d))
> (list-length '(b c d))
> >(list-length '(c d))
> > (list-length '(d))
> > >(list-length '())
< < <0
< < 1
< <2
< 3
<4
4
```

Tail recursion

```
(define list-length-tail
  (lambda (lst result)
    (if (null? lst) result
        (list-length-tail (cdr lst) (+ 1 result)))))
(trace list-length-tail)
(list-length-tail '(a b c d) 0)
```

```
>(list-length-tail '(a b c d) 0)
>(list-length-tail '(b c d) 1)
>(list-length-tail '(c d) 2)
>(list-length-tail '(d) 3)
>(list-length-tail '() 4)
<4
4
```

- Note an extra parameter is required.
- This can be eliminated with a front-end function

Nontail copy vs. tail copy

```
(define list-copy
  (lambda (lst)
    (if (null? lst) '()
        (cons (car lst)
              (list-copy (cdr lst))))))
                         result)))))
```

```
>(list-copy '(a b c d))
> (list-copy '(b c d))
> >(list-copy '(c d))
> > (list-copy '(d))
> > >(list-copy '())
< < <'()
< < '(d)
< <'(c d)
< '(b c d)
<'(a b c d)
'(a b c d)
```

Nontail copy vs. tail copy

```
(define list-copy-tail
  (lambda (lst result)
    (if (null? lst) (reverse result)
        (list-copy-tail (cdr lst) (cons (car lst)
                                        result)))))
```

```
>(list-copy-tail '(a b c d) '())
>(list-copy-tail '(b c d) '(a))
>(list-copy-tail '(c d) '(b a))
>(list-copy-tail '(d) '(c b a))
>(list-copy-tail '() '(d c b a))
<'(a b c d)
'(a b c d)
```

Tree recursion

```
(define tree-copy (lambda (tree)
    (if (not (pair? tree)) tree
        (cons (tree-copy (car tree)) (tree-copy (cdr
  tree))))))
```

```
>(tree—copy '((a) (b)))
> (tree—copy '(a))
> >(tree—copy 'a)
< <'a
> >(tree—copy '())
< <'()
< '(a)
> (tree—copy '((b)))
> >(tree—copy '(b))
> > (tree—copy 'b)
< < 'b
> > (tree—copy '())
< < '()
< <'(b)
> >(tree—copy '())
< <'()
< '((b))
 <'((a) (b))
```

- Tail recursion not possible.
- Not a linear process

Named `let`

- Named function:

```
(define fib
  (lambda (n)
    (if (< n 2) 1
        (+ (fib (- n 2))
           (fib (- n 1))))))
(fib 10)
```

- Named `let`

```
(let fib ((n 10))
  (if (< n 2) 1
      (+ (fib (- n 2))
         (fib (- n 1)))))
```

Local recursive functions

```scheme
(define mod2
  (lambda (n)
    (letrec ((even?
               (lambda (n)
                 (if (zero? n)
                     #t
                     (odd? (- n 1)))))
              (odd?
               (lambda (n)
                 (if (zero? n)
                     #f
                     (even? (- n 1))))))
  (cond ((even? n) 0)
        ((odd? n) 1)
        (else 0)))))
```

```scheme
> (mod2 2341)
1
```

Local recursive functions

```
(define (mod2 n)
  [local
    [(define (even? n)
       (if (zero? n)
           #t
           (odd? (- n 1))))
     (define (odd? n)
       (if (zero? n)
           #f
           (even? (- n 1))))]
    (cond ((even? n) 0)
          ((odd? n) 1)
          (else 0))])
```

```
> (mod2 2341)
1
>
```

define-type from PLAI

```
#lang plai

(define-type foo
  (bar (x number?))
  (mug (x string?)))

(define (double x)
  (type-case foo x
    (bar (x) (* 2 x))
    (mug (x) (string-append x x))))

(let ((myfoo (bar 99))
      (nofoo (mug "hello")))
  (list (double myfoo)
        (double nofoo)))
```

define-type from PLAI

```
#lang plai
(define-type position
  (2d (x number?) (y number?))
  (3d (x number?) (y number?) (z number?)))

(define-type shape
  (circle (center 2d?)
          (radius number?))
  (square (lower-left 2d?)
          (width number?)
          (height number?)))
```

```
(define (area s)
  (type-case shape s
    (circle (center radius)
            (* pi radius radius))
    (square (lower-left width height)
            (* width height))))
```

Center of a shape

```
(define (center s)
  (type-case shape s
    (circle (center radius) center)
    (square (lower-left width height)
            (type-case position lower-left
              (2d (x y)
                  (2d (+ x (/ width 2))
                      (+ y (/ height 2))))
              (else (error "bad square" s))))))
```

Returning multiple values

```
(define foo
  (lambda (x) (values x (* x x))))

(define bar
  (lambda (x) (values x (* x x x))))

(let-values (((a b) (foo 3))
             ((c d) (bar 3)))
  (list a b d))
```

Unit testing

```
#lang plai

(define slow+
  (lambda (a b)
    (if (zero? a) b
        (slow+ (sub1 a)
               (add1 b)))))

(define slow*
  (lambda (a b)
    (if (zero? a) 0
        (slow+ b
          (slow*
            (sub1 a) b))))))
```

arithmetic.rkt

```
#lang plai

(require rackunit
         "arithmetic.rkt")

;; These provided by plai:
(test (slow+ 4 5) 9)
(test (slow* 4 5) 21)

;; These by rackunit:
(check-equal?
  (slow+ 4 5) 9)
(check-equal?
  (slow* 4 5) 21)
```

arithmetic-test.rkt

Homework 2: Programming Scheme

- Programming puzzles in Scheme.
- Must use **natural recursion** and not builtin Scheme functionality.
- Review `naturalrecursion.rkt`