

Intro to Scheme

Geoffrey Matthews

January 11, 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

```
1 > 123
2 123
3 > 1.2e5
4 120000.0
5 > 1/3
6 1/3
7 > 'a
8 'a
9 > 'very-long-identifier
10 'very-long-identifier
11 > '(list of things 1 2 3)
12 '(list of things 1 2 3)
```

```
1 > "a string"
2 "a string"
3 > ")) a bad string"
4 ")) a bad string"
5 > '99
6 99
7 > 99
8 99
9 > #t
10 #t
11 > '#t
12 #t
```

Basic Scheme Syntax: Procedure calls

```
1 > (+ 3 4)
2 7
3 > (+ 3 (* 4 5))
4 23
5 > (* (+ 3 4) 5)
6 35
7 > ((if #f + *) 3 4)
8 12
9 >
```

```
1 3 + 4 * 5
```

Basic Scheme Syntax: Naming things

```
1 > (define x 2)
2 > (define y 3)
3 > x
4 2
5 > y
6 3
7 > (* x y)
8 6
```

Basic Scheme Syntax: Conditionals

```
1 > (define x 2)
2 > (define y 3)
3 > (if (< x y) x y)
4 2
5 > (cond ((< x y) 'yes)
6         ((> x y) 'no)
7         ((= x y) 'maybe)
8         (else (error "Help!" (list x y))))
9 'yes
10 >
```

Basic Scheme Syntax: Case

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

Printing: three ways

```
1 (define astring
2   "foobuz
3   basssssssssssssss @#$@#$")
4 (display astring)
5 (newline)
6 (print astring)
7 (newline) ;; Notice neither prints a newline by itself
8 (printf
9   "Formatted output: ~a and: ~a\n"
10  'aThing 1234)
```

```
1 foobuz
2 basssssssssssssss @#$@#$
3 "foobuz\nbasssssssssssssss @#$@#$"
4 Formatted output: aThing and: 1234
```


Basic Scheme Syntax: Sequencing

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

Two ways to define procedures

```
1 (define (square1 x) (* x x))
2 (define square2 (lambda (x) (* x x)))
3 ;; We will use the second way.
4
5 ;; simple unit tests:
6 (define testnumber 23423)
7 (test (square1 testnumber)
8       (square2 testnumber))
```

```
1 (good (square1 testnumber) 548636929 548636929 "at
   line 31")
```

Rational numbers and many other builtin types

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

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

Two ways to quote things

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

```
1 (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.

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

```
1 (good '(1 2 3) '(1 2 3) '(1 2 3) "at line 49")
2 (good (cons 3 (cons 2 (cons 1 '()))) '(3 2 1) '(3 2 1)
3      "at line 52")
4 '((a . b) (c (d . e) (f g)))
5 '(1 (2 (3 (4 5))))
6 '(1 2 3 4 . 5)
```

car's and cdr's can be smashed together

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

let introduces local variables

```
1 (let ((x 3)
2       (y 4))
3     (test (* x y)
4            (* 3 4)))
5
6 ;; This doesn't work:
7
8 ;; (let ((x 3)
9 ;;       (y (* 5 x)))
10 ;;      (* x y))
```

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

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


Shadowed variables

```
1 (let ((b 99999))
2   (let ((a 100)
3         (b 1000)
4         (c 10000))
5     (+ a b c)))
6
7 (let ((b 99999))
8   (+ (let ((a 100)
9         (b 1000)
10        (c 10000))
11       (+ a b c))
12      b))
```

Functions are first class values

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

Functions are first class values

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

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

lambda and let are similar

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

Free variables can be captured by lambda

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

- Free variables remain captured even when shadowed:

```
1 (let ((x 'sam))
2   (let ((f (lambda (y z) (list x y z))))
3     (let ((x 'mary))
4       (f 'i 'am))))
```

lambda creates a **closure**

```
1 (define f1
2   (let ((x 'sam))
3     (lambda (y z) (list x y z))))
4
5 (f1 'i 'am)
6 (define x 'nobody)
7 (f1 'i 'am)
```

```
1 '(sam i am)
2 '(sam i am)
```

We can make objects with local state

```
1 (define counter
2   (let ((n 0))
3     (lambda ()
4       (set! n (+ n 1))
5       n)))
6
7 (counter)
8 (counter)
9 (counter)
10 (counter)
```

```
1 1
2 2
3 3
4 4
```

Functions can return functions

```
1 (define adder
2   (lambda (x)
3     (lambda (y) (+ x y))))
4
5 (define f (adder 10))
6 (define g (adder 100))
7 (list (f 3) (f 5) (g 3) (g 5))
```

```
1 '(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

```
1 (define triple
2   (lambda (a b c)
3     (lambda (op)
4       (cond ((eqv? op 'first) a)
5              ((eqv? op 'second) b)
6              ((eqv? op 'third) c)))))
7
8 (define a (triple 5 9 20))
9 (define b (triple 'hello 'goodbye 'whatever))
10 (list (a 'first) (b 'second) (a 'third) (b 'first))
```

```
1 '(5 goodbye 20 hello)
```

We can use closures to create objects like stacks

```
1 (define stack
2   (lambda ()
3     (let ((the-stack '()))
4       (lambda (op . args)
5         (cond ((eq? op 'push)
6                 (set! the-stack (cons (car args)
7                                         the-stack)))
7               ((eq? op 'pop)
8                 (let ((top (car the-stack)))
9                   (set! the-stack (cdr the-stack))
10                     top)))
11              (else
12                (error "Unknown stack operator: ~a"
13                        op)))))))
14
15 (define s (stack))
16 (s 'push 99)
17 (s 'push 101)
18 (list (s 'pop) (s 'pop))
```

```
1 '(101 99)
```

List recursion

```
1 (require racket/trace)
2 (define list-length
3   (lambda (lst)
4     (if (null? lst) 0
5         (+ 1 (list-length (cdr lst))))))
6 (trace list-length)
7 (list-length '(a b c d))
```

```
1 (list-length '(a b c d))
2 > (list-length '(b c d))
3 > >(list-length '(c d))
4 > > (list-length '(d))
5 > > >(list-length '())
6 < < <0
7 < < 1
8 < <2
9 < 3
10 <4
11 4
```

Tail recursion

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

```
1 >(list-length-tail '(a b c d) 0)
2 >(list-length-tail '(b c d) 1)
3 >(list-length-tail '(c d) 2)
4 >(list-length-tail '(d) 3)
5 >(list-length-tail '() 4)
6 <4
7 4
```

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

Nontail copy vs. tail copy

```
1 (define list-copy
2   (lambda (lst)
3     (if (null? lst) '()
4         (cons (car lst)
5               (list-copy (cdr lst))))))
6   result)))))
```

```
1 >(list-copy '(a b c d))
2 > (list-copy '(b c d))
3 > >(list-copy '(c d))
4 > > (list-copy '(d))
5 > > >(list-copy '())
6 < < <'()
7 < < '(d)
8 < <'(c d)
9 < '(b c d)
10 <'(a b c d)
11 '(a b c d)
```

Nontail copy vs. tail copy

```
1 (define list-copy-tail
2   (lambda (lst result)
3     (if (null? lst) (reverse result)
4         (list-copy-tail (cdr lst) (cons (car lst)
5                                           result)))))
```

```
1 >(list-copy-tail '(a b c d) '())
2 >(list-copy-tail '(b c d) '(a))
3 >(list-copy-tail '(c d) '(b a))
4 >(list-copy-tail '(d) '(c b a))
5 >(list-copy-tail '() '(d c b a))
6 <'(a b c d)
7 '(a b c d)
```

Tree recursion

```
1 (define tree-copy (lambda (tree)
2   (if (not (pair? tree)) tree
3       (cons (tree-copy (car tree)) (tree-copy (cdr
4         tree))))))
```

```
1 >(tree-copy '((a) (b)))
2 > (tree-copy '(a))
3 > >(tree-copy 'a)
4 < <'a
5 > >(tree-copy '())
6 < <'()
7 < '(a)
8 > (tree-copy '((b)))
9 > >(tree-copy '(b))
10 > > (tree-copy 'b)
11 < < 'b
12 > > (tree-copy '())
13 < < '()
14 < <'(b)
15 > >(tree-copy '())
16 < <'()
17 < '((b))
18 < '((a) (b))
```

- Tail recursion not possible.
- Not a linear process

Named let

- Named function:

```
1 (define fib
2   (lambda (n)
3     (if (< n 2) 1
4         (+ (fib (- n 2))
5             (fib (- n 1))))))
6 (fib 10)
```

- Named let

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


Local recursive functions

```
1 (define mod2
2   (lambda (n)
3     (letrec ((even?
4               (lambda (n)
5                 (if (zero? n)
6                     #t
7                     (odd? (- n 1)))))
8               (odd?
9                 (lambda (n)
10                  (if (zero? n)
11                      #f
12                      (even? (- n 1)))))))
13   (cond ((even? n) 0)
14         ((odd? n) 1)
15         (else 0))))
16 > (mod2 2341)
17 1
```

Local recursive functions

```
1 > (define mod2
2   (lambda (n)
3     (define even?
4       (lambda (n)
5         (if (zero? n)
6             #t
7             (odd? (- n 1)))))
8     (define odd?
9       (lambda (n)
10        (if (zero? n)
11            #f
12            (even? (- n 1)))))
13    (cond ((even? n) 0)
14          ((odd? n) 1)
15          (else 0)))
16 > (mod2 2341)
17 1
18 >
```

define-type from PLAI

```
1 #lang plai
2
3 (define-type foo
4   (bar (x number?))
5   (mug (x string?)))
6
7 (define (double x)
8   (type-case foo x
9     (bar (x) (* 2 x))
10    (mug (x) (string-append x x))))
11
12 (let ((myfoo (bar 99))
13       (nofoo (mug "hello")))
14   (list (double myfoo)
15         (double nofoo)))
```

define-type from PLAI

```
1 #lang plai
2 (define-type position
3   (2d (x number?) (y number?))
4   (3d (x number?) (y number?) (z number?)))
5
6 (define-type shape
7   (circle (center 2d?)
8           (radius number?))
9   (square (lower-left 2d?)
10          (width number?)
11          (height number?)))
```

```
1 (define (area s)
2   (type-case shape s
3     (circle (center radius)
4             (* pi radius radius))
5     (square (lower-left width height)
6             (* width height))))
```

Center of a shape

```
1 (define (center s)
2   (type-case shape s
3     (circle (center radius) center)
4     (square (lower-left width height)
5              (type-case position lower-left
6                (2d (x y)
7                     (2d (+ x (/ width 2))
8                          (+ y (/ height 2))))
9                (else (error "bad square" s))))))
```

Returning multiple values

```
1 (define foo
2   (lambda (x) (values x (* x x))))
3
4 (define bar
5   (lambda (x) (values x (* x x x))))
6
7 (let-values (((a b) (foo 3))
8              ((c d) (bar 3)))
9   (list a b d))
```

Unit testing

```
1 #lang plai
2
3 (define slow+
4   (lambda (a b)
5     (if (zero? a) b
6         (slow+ (sub1 a)
7                 (add1 b)))))
8
9 (define slow*
10  (lambda (a b)
11    (if (zero? a) 0
12        (slow+ b
13                (slow*
14                  (sub1 a) b)))))
```

arithmetic.rkt

```
1 #lang plai
2
3 (require rackunit
4        "arithmetic.rkt")
5
6 ;; These provided by plai:
7 (test (slow+ 4 5) 9)
8 (test (slow* 4 5) 21)
9
10 ;; These by rackunit:
11 (check-equal?
12   (slow+ 4 5) 9)
13 (check-equal?
14   (slow* 4 5) 21)
```

arithmetic-test.rkt