Scheme Notes 01

Geoffrey Matthews

Department of Computer Science
Western Washington University

September 25, 2018

Resources

- The software:
  - `https://racket-lang.org/`
- Texts:
  - `https://mitpress.mit.edu/sicp/`
  - `http://www.scheme.com/tspl3/`
    (make sure you use the 3rd edition and not the 4th)
  - `http://ds26gte.github.io/tyscheme/`

Running the textbook examples

- Using the racket language is usually best, the examples from *The Scheme Programming Language* should run without modification.
- The examples from SICP are a little more idiosyncratic. Most of them can be run by installing the `sicp` package as in these instructions:
  `http://stackoverflow.com/questions/19546115/which-lang-packet-is-proper-for-sicp-in-dr-racket`

Simple Scheme Program

quadratic.rkt

```
(provide quadratic)

(define quadratic
  (lambda (a b c)
    (let ((discriminant (- (* b b) (* 4.0 a c))))
      (if (< discriminant 0.0)
          (list +nan.0 +nan.0)
          (two-real-solutions (- b)
                              (sqrt discriminant)
                              (* 2.0 a))))))

(define two-real-solutions
  (lambda (neg-b root-disc two-a)
    (list (/ (- neg-b root-disc) two-a)
          (/ (+ neg-b root-disc) two-a))))
```

Simple Scheme Program Unit Tests

quadratic-test.rkt

```
(define same?
  (lambda (a b)
    (< (abs (- a b)) 1.0e-10)))

(define list-same?
  (lambda (ls1 ls2)
    (and (same? (first ls1) (first ls2))
         (same? (second ls1) (second ls2)))))

(check list-same?
       (quadratic 1 2 1) (list -1.0 -1.0))
(check list-same?
       (quadratic 1 0 -1) (list -1.0 1.0))
(check list-same?
       (quadratic 1 -5 6) (list 2.0 3.0))
```

There are two types of expressions:

- **Primitive expressions:** no parentheses.
  Constants:
  `4    3.141592    #t    #f    "Hello world!"`
  Variables:
  `x    long-variable-name    a21`
- **Compound expressions:** with parentheses.
  Special forms:
  ```
  (define x 99)
  (if (> x y) x y)
  ```
  Function calls:
  ```
  (+ 1 2 3)
  (list (list 1 2) (list 3 4))
  ```

Introducing Local Variables

```
(let ((x 3)
      (y 4)
      (z 5))
    (+ x (* y z)))    =>  23
```

Beware! This will NOT work.

```
(let ((x 3)
      (y (* 2 x))
      (z (* 3 x)))
    (+ x (* y z)))    =>  57
```

But this will.

```
(let* ((x 3)
       (y (* 2 x))
       (z (* 3 x)))
    (+ x (* y z)))    =>  57
```

Defining Procedures

Two equivalent ways:

```
(define (square x) (* x x))
(define square (lambda (x) (* x x)))
```

The first one is more in line with the procedure call:

```
(square 5) => 25
```

Defining Procedures

Two equivalent ways:

```
(define (square x) (* x x))
(define square (lambda (x) (* x x)))
```

The second one is more in line with defining other things:

```
(define x (* 3 4))
(define y (list 5 9 22))
```

The action of `define` is simply to give a *name* to the result of an expression.

Defining Procedures

Two equivalent ways:

```
(define (square x) (* x x))
(define square (lambda (x) (* x x)))
```

The result of a lambda-expression is an anonymous function.
We can name it, as above, or use it without any name at all:

```
(square 5) => 25
((lambda (x) (* x x)) 5) => 25
```

Procedures always return a value

```
(define (bigger a b c d)
  (if (> a b) c d))
```

```
(define (solve-quadratic-equation a b c)
  (let ((disc (sqrt (- (* b b)
                        (* 4.0 a c)))))
    (list
     (/ (+ b disc)
        (* 2.0 a))
     (/ (+ (- b) disc)
        (* 2.0 a)))
    ))
```

Solving problems

Newton's method:
If y is a guess for $\sqrt{x}$, then the average of y and x/y is an even better guess.

x	guess	quotient	average
2	1.0	2.0	1.5
2	1.5	1.3333333333333333	1.4166666666666665
2	1.4166666666666665	1.411764705882353	1.4142156862745097
2	1.4142156862745097	1.41421143847487	1.4142135623746899

...

Evidently, we want to iterate, and keep recomputing these things until we find a value that's close enough.

Newton's Method in Scheme

```
(define sqrt-iter
  (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x))))

(define improve
  (lambda (guess x)
    (average guess (/ x guess))))

(define average
  (lambda (x y) (/ (+ x y) 2)))

(define good-enough?
  (lambda (guess x)
    (< (abs (- (square guess) x)) 0.00001)))

(define square
  (lambda (x) (* x x)))

(define sqrt
  (lambda (x) (sqrt-iter 1.0 x)))
```

Decompose big problems into smaller problems.

Newton's Method in Scheme

```
(define sqrt-iter
  (lambda (guess x)
    (if (good-enough? guess x)
        guess
        (sqrt-iter (improve guess x) x))))

(define improve
  (lambda (guess x)
    (average guess (/ x guess))))

(define average
  (lambda (x y) (/ (+ x y) 2)))

(define good-enough?
  (lambda (guess x)
    (< (abs (- (square guess) x)) 0.00001)))

(define square
  (lambda (x) (* x x)))

(define sqrt
  (lambda (x) (sqrt-iter 1.0 x)))
```

Note: NO GLOBAL VARIABLES!

Definitions can be nested

```
(define sqrt
  (lambda (x)
    (define good-enough?
      (lambda (guess x)
        (< (abs (- (square guess) x)) 0.001)))
    (define improve
      (lambda (guess x)
        (average guess (/ x guess))))
    (define sqrt-iter
      (lambda (guess x)
        (if (good-enough? guess x)
            guess
            (sqrt-iter (improve guess x) x))))
    (sqrt-iter 1.0 x)))
```

Parameters need not be repeated

```
(define sqrt
  (lambda (x)
    (define good-enough?
      (lambda (guess)
        (< (abs (- (square guess) x)) 0.001)))
    (define improve
      (lambda (guess)
        (average guess (/ x guess))))
    (define sqrt-iter
      (lambda (guess)
        (if (good-enough? guess)
            guess
            (sqrt-iter (improve guess)))))
    (sqrt-iter 1.0)))
```

Introducing local functions with letrec

```
(define sqrt
  (lambda (x)
    (letrec ((good-enough?
               (lambda (guess)
                 (< (abs (- (square guess) x)) 0.001)))
              (improve
               (lambda (guess)
                 (average guess (/ x guess))))
              (sqrt-iter
               (lambda (guess)
                 (if (good-enough? guess)
                     guess
                     (sqrt-iter (improve guess)))))
              )
              (sqrt-iter 1.0))))
```

Procedures in Scheme

```
(define f
  (lambda (x) (* x x))
  )
```

```
(f 3) => 9
(f 4) => 16

((lambda (x) (* x x)) 3) => 9
((lambda (x) (* x x)) 4) => 16
```

Local Variables in Procedures: Closures

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

(counter)   =>  1
(counter)   =>  2
(counter)   =>  3
```

A Procedure that Returns a Procedure

```
(define curry
  (lambda (a)
    (lambda (b)
      (+ a b)
    )
  )
)
```

```
(curry 3)         =>  #<procedure>

((curry 3) 4)     =>  7
```

A Procedure that Returns a Counter

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

(define x (new-counter))
(define y (new-counter))

(x)  => 1
(x)  => 2
(x)  => 3
(y)  => 1
(y)  => 2
(x)  => 4
```

Procedures as Returned Values: Derivatives

```
(define d
  (lambda (f)
    (let* ((delta 0.00001)
           (two-delta (* 2 delta)))
      (lambda (x)
        (/ (- (f (+ x delta)) (f (- x delta)))
           two-delta)))))
```

```
((d (lambda (x) (* x x x)))  5)     => ?
((d sin) 0.0)                       => ?
```

Procedures as Returned Values: Procedures as Data

```
(define make-pair
  (lambda (a b)
    (lambda (i)
      (cond ((= i 0) a)
            ((= i 1) b)
            (else (error 'bad-index))))
  ))

(define x (make-pair 4 8))
(define y (make-pair 100 200))
(define z (make-pair x y))

(x 0)         => ?
(y 1)         => ?
((z 1) 0)     => ?
```

Procedures as parameters

Summation notation:

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

In scheme:

```
(define sum
  (lambda (a b f)
    (if (> a b)
        0
        (+ (f a) (sum (+ a 1) b f)))))

(sum 1 10 square) => 385
(sum 1 10 (lambda (x) (* x x x))) => 3025
```

Better notation for summations

Instead of

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

use

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

Why don't we use that?

Better notation for summations

Instead of

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

use

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

Why don't we use that?

Because then you have problems with

$$\sum_{i=a}^{b} i^2 = a^2 + \ldots + b^2$$

$$\sum_{a}^{b} \boxed{?} = a^2 + \ldots + b^2$$

Better notation for summations

Instead of

$$\sum_{i=a}^{b} f(i) = f(a) + \ldots + f(b)$$

use

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

Why don't we use that?

Because then you have problems with

$$\sum_{i=a}^{b} i^2 = a^2 + \ldots + b^2$$

$$\sum_{a}^{b} \boxed{?} = a^2 + \ldots + b^2$$

$$\sum_{a}^{b} \lambda i.i^2 = a^2 + \ldots + b^2$$

Matches Scheme Code Better, Too

$$\sum_{a}^{b} f = f(a) + \ldots + f(b)$$

```
(define sum
  (lambda (a b f)
    (if (> a b)
        0
        (+ (f a) (sum (+ a 1) b f)))))
```

Finding fixed points

x is a *fixed point* of f if $x = f(x)$
For some functions you can find fixed points by iterating:
$x, f(x), f(f(x)), f(f(f(x))), \ldots$

Fixed points in scheme:

```
(define fixed-point
  (lambda (f)
    (let
        ((tolerance 0.0001)
         (max-iterations 10000))
      (letrec
          ((close-enough?
            (lambda (a b) (< (abs (- a b)) tolerance)))
           (try
            (lambda (guess iterations)
              (let ((next (f guess)))
                (cond ((close-enough? guess next) next)
                      ((> iterations max-iterations) #f)
                      (else (try next (+ iterations 1))))))))
           )
        (try 1.0 0)))))

(fixed-point cos)  => 0.7390547907469174
(fixed-point sin)  => 0.08420937654137994
(fixed-point (lambda (x) x)) => 1.0
(fixed-point (lambda (x) (+ x 1))) => #f
```

Remember Newton's Method?

Newton's method:
If y is a guess for $\sqrt{x}$, then the average of y and x/y is an even better guess.

x	guess	quotient	average
2	1.0	2.0	1.5
2	1.5	1.3333333333333333	1.4166666666666665
2	1.4166666666666665	1.411764705882353	1.4142156862745097
2	1.4142156862745097	1.41421143847487	1.4142135623746899
...			

Evidently, we want to iterate, and keep recomputing these things until we find a value that's close enough.

Newton's Method Using Functional Programming

```
(define sqrt
  (lambda (x)
    (fixed-point
      (lambda (y)
        (/
           (+ y (/ x y))
           2
        )))))
```