Scheme Notes 01

Geoffrey Matthews

Department of Computer Science
Western Washington University

January 12, 2017

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`

Every powerful language has

- primitive expressions: the simplest entities, such as 3 and +
- means of combination: buidling compound elements from simpler ones such as
  `(+ 3 4)`
  - In Scheme combinations are always parentheses, with the operator first and the operands following.
- means of abstraction: a way for naming compound elements and then manipulating them as units such as

```
(define pi 3.14159)
(define square (lambda (x) (* x x)))
```

The REPL does the following three things:

- Reads an expression
- Evaluates it to produce a value
- Prints the value

The returned value has a small set of types, including number, boolean and procedure. (Later, we'll see symbol, pair, vector, and promise (stream).)

There are 4 types of expressions:

- Constants: numbers, booleans. Examples: `4 3.141592 #t #f`
- Variables: names for values. We create these using the special form `define`
- Special forms: have special rules for evaluation. In addition, you may not redefine a special form.
- Combinations: `(<operator> <operands>)`. These are sometimes called "function calls" or "procedure applications."

The first two types of expressions (constants and variables) are primitive expressions – they have no parentheses. The second two types are called compound expressions – they have parentheses.

Mantras

- Every expression has a value
  - (except for errors, infinite loops and the `define` special form)
- To find the value of a combination:
  - Find values of all subexpressions in any order
  - Apply the value of the first to the values of the rest
- The value of a `lambda` expression is a procedure

Finding the value of a combination

- Find values of all subexpressions in any order
- Apply the value of the first to the values of the rest

$$\texttt{(+ (* 2 3) }\boxed{\texttt{(- 8 2)}}\texttt{)}$$

$$\texttt{(+ }\boxed{\texttt{(* 2 3)}}\texttt{ 6)}$$

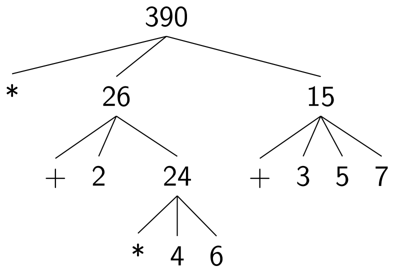
$$\boxed{\texttt{(+ 6 6)}}$$

$$\texttt{12}$$

Program Evaluation In Lisp

A process of *tree accumulation.*

```
(* (+ 2 (* 4 6))
   (+ 3 5 7))
```



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 'a 'b 'c))
```

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.

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 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?
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

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

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

We have the same problem with derivatives

$$\frac{dx^2}{dx} = 2x$$

$$D(\lambda x.x^2) = \lambda x.2x$$

The chain rule

With pure functions it's easy:

$$D(f \circ g) = (D(f) \circ g) \cdot D(g)$$

With applied functions:

$$\begin{aligned} F &= f \circ g \\ F(x) &= f(g(x)) \\ F'(x) &= f'(g(x))g'(x) \\ \frac{dF(x)}{dx} &= \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx} \end{aligned}$$

Or, if we let $z = f(y)$ and $y = g(x)$ (which is weird) then it looks kind of nice and is easy to memorize (cancel fractions!):

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

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?

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

Procedures as Returned Values

```
(define make-adder
  (lambda (n)
    (lambda (m) (+ m n))))
```

```
((make-adder 4) 5)
```

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)
```

Procedures as Returned Values: Procedures as Data

```
(define make-pair
  (lambda (a b)
    (lambda (command)
      (if (eq? command 'first) a b))))

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

Procedures as Returned Values

```
(define average-damp
  (lambda (f)
    (lambda (x) (/ (+ x (f x)) 2))))

(define sqrt
  (lambda (x)
    (fixed-point (average-damp (lambda (y) (/ x y))))))
```