Scheme Notes 01

Geoffrey Matthews

Department of Computer Science
Western Washington University

January 8, 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))
```

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.
  - `if` and `define` are special forms.
- 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

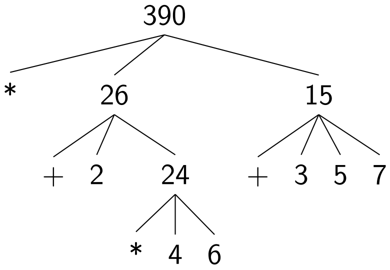
- Find values of all subexpressions in any order
- Apply the value of the first to the values of the rest

```
(+ (* 2 3) [(- 8 2)])

(+ [(* 2 3)]    6    )

[(+    6      6    )]

       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 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))))
```