```
;;;; 6.945, 6.905 - Quick Intro to Scheme
;;;              2/5/2016

#|
Why Scheme/Lisp?
 - Easy to demonstrate ideas without "ceremony"
 - Almost everything is redefinable
 - Almost no syntax. Basically just parse tree
   => easy to write programs
        that read/write programs
|#
```

```
;;; --------------------------
;;; Primitives
;;; --------------------------

;;; Numerals -- represent numbers

1
;Value: 1

2.5
;Value: 2.5

3/4
;Value: 3/4

3+4i
;Value: 3+4i

;;; Procedures
+
;Value: #[arity-dispatched-procedure 17 +]

*
;Value: #[arity-dispatched-procedure 18 *]
```

```
;;; --------------------------
;;; Combinations
;;; --------------------------

;;; Procedure call:
;;; (operator operand-1 ... operand-n)

(+ 1 3/4)
;Value: 7/4

(+ 1 2 3)
;Value: 6

(* 4 (+ 1 2 3) 5)
;Value: 120

(- 3 4)
;Value: -1

(- 3)
;Value: -3

(/ (* 4 (+ 1 2 3)) 5)
;Value: 24/5
```

```
;;; Literal Procedures

(lambda (x) (* x x))

;;; Procedure expression:
;;; (lambda <formal-parameters> <body>)
;;; Can be used anywhere we need a procedure:

((lambda (x) (* x x)) 3)
;Value: 9

;;; When called, evaluates to value of body
;;; with arguments substituted for
;;; parameters.


;;; --------------------------
;;; Abstractions
;;; --------------------------

;;; Can give a name to any object:

(atan 1 1)
;Value: .7853981633974483

(define pi (* 4 (atan 1 1)))
;Value: pi

pi
;Value: 3.141592653589793
```

```
(define square
  (lambda (x) (* x x)))
;Value: square

(square 3)
;Value: 9

(define area-circle
  (lambda (r)
    (* pi (square r))))

(area-circle 3)
;Value: 28.274333882308138

(define (area-circle r)
  (* pi (square r)))

(define (area-ellipse semimajor semiminor)
  (* pi semimajor semiminor))

(define (cube x) (* x x x))
```

```
;;; -------------------------
;;; Local Variables
;;; -------------------------

;;; Name objects in local context

;;; General form:
;;; (let ((variable-1 expression-1)
;;;       ...
;;;       (variable-n expression-n))
;;;   body)

(define (f radius)
  (let ((area (* 4 pi (square radius)))
        (volume (* 4/3 pi (cube radius))))
    (/ volume area)))

(f 1)
;Value: .3333333333333333

(f 2)
;Value: .6666666666666666

(f 3)
;Value: .9999999999999999
```

```
;;; --------------------------
;;; Conditional Expressions
;;; --------------------------

;;; General form:
;;; (cond (predicate-1 consequent-1)
;;;       ...
;;;       (predicate-n consequent-n))

(define (abs x)
  (cond ((< x 0) (- x))
        ((= x 0) 0)
        ((> x 0) x)))


;;; Else clause

(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))


;; If expression

(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

```
;;; -------------------------
;;; Recursive/Iterative Procedures
;;; -------------------------

;;; Recursive Procedure

(define (^ x p)
  (if (= p 0)
      1
      (* x (^ x (- p 1)))))

(^ 3 4)
;Value: 81

;;; Iterative Procedure

(define (^ x p)
  (define (iter count product)
    (if (= count 0)
        product
        (iter (- count 1)
              (* x product))))
  (iter p 1))

;;; Let variant Syntax
(define (^ x p)
  (let iter ((count p) (product 1))
    (if (= count 0)
        product
        (iter (- count 1)
            (* x product)))))
```

```
;;; --------------------------
;;; General exponentiation
;;; --------------------------

(define (^ x p)
  (define (lp p)
    (cond ((= p 1) x)
          ((even? p)
           (square (lp (/ p 2))))
          ((odd? p)
           (* x (lp (- p 1))))))
  (if (exact-integer? p)
      (cond ((= p 0) 1)       ;0^0 -> 1
            ((< p 0) (^ (/ 1 x) (- p)))
            (else (lp p)))
      (exp (* p (log x)))))

;;; For fun: This was done by recursive
;;;  repeated squaring.  Write the
;;;  iterative version of this algorithm.
```

```
;;; --------------------------
;;; First-class functions
;;; --------------------------

(define compose
  (lambda (f g)
    (lambda (x)
      (f (g x)))))

;;; Alternatively, could write
(define (compose f g)
  (lambda (x)
    (f (g x))))

;;; Or even
(define ((compose f g) x)
  (f (g x)))

(define (identity x) x)

(define (repeated f n)
  (if (= n 0)
      identity
      (compose f (repeated f (- n 1)))))

((repeated square 3) 2)
;Value: 256
```

```
;;; -------------------------
;;; Compound Data - Lists
;;; -------------------------

(define factors-of-28
  (list 1 2 4 7 14 28))

factors-of-28
;Value: (1 2 4 7 14 28)

(pair? factors-of-28)
;Value: #t

(car factors-of-28)
;Value: 1

(cdr factors-of-28)
;Value: (2 4 7 14 28)

(car (cdr factors-of-28))
;Value: 2

;;; cadr cdar caar cadar
(cadr factors-of-28)
;Value: 2
```

```
;;; --------------------------
;;; Operations on Lists
;;; --------------------------

(define (addup lst)
  (if (null? lst)
      0
      (+ (car lst)
         (addup (cdr lst)))))

(/ (addup factors-of-28) 2)
;Value: 28
;;; 28 is a "perfect number".

;;; Last element is empty list,
;;; identified by null? predicate
(cdr (cdr (list 1 2)))
;Value: ()

(null? (cdr (cdr (list 1 2))))
;Value: #t
```

```
;;; --------------------------
;;; Compound Data - Pairs
;;; --------------------------

;;; Pairs are made by cons
(cons 1 2)
;Value: (1 . 2)

(define p (cons 1 2))
;Value: p

(car p)
;Value: 1

(cdr p)
;Value: 2

;;; A list is a string of cons pairs
;;;  ending in the empty list.
```

```
;;; -------------------------
;;; Map, Filter, Reduce
;;; -------------------------

;;; Lists are chains of pairs.

(define (map f lst)
  ;; In standard library
  (if (null? lst) ; Test for the empty list
      '()         ; What is this "'()"?
      (cons (f (car lst))
            (map f (cdr lst)))))

(map square factors-of-28)
;Value: (1 4 16 49 196 784)

(filter odd?
        (map square factors-of-28))
;Value: (1 49)

(reduce +
        0
        (filter odd?
                (map square factors-of-28)))
;Value: 50
```

```
;;; --------------------------
;;; Quotation and Symbols
;;; --------------------------

;;; Scheme expressions look like lists
;;; but can contain symbols. Need a way
;;; to refer to those symbols in code:
;;; -> Symbols and quotation.

;;; Quotation:
(define my-list '(1 2 3))
(define my-list (list 1 2 3))

;;; Symbols:
;;; Primitive data type

;;; These pairs are equivalent:

(define symbolic-sum '(+ 2 3))
(define symbolic-sum (list '+ 2 3))

(define symbolic-define
  '(define pi 3.14))

;;; Or
(define symbolic-define
  (list 'define 'pi 3.14))

;;; eq? tests for identical symbols
(eq? (car symbolic-define) 'define)
;Value: #t
```

```
;;; -------------------------
;;; Programs are Lists
;;; -------------------------

;;; With quote, program becomes list:
(define foo
  '(define compose
     (lambda (f g)
       (lambda (x)
         (f (g x))))))

;;; What does mangle do?

(define (mangle expr)
  (if (and (eq? (car expr) 'define)
           (pair? (caddr expr))
           (eq? (caaddr expr) 'lambda))
      (mangle
       (cons 'define
             (cons (cons (cadr expr)
                         (cadr (caddr expr)))
                   (cddr (caddr expr)))))
      expr))

(mangle foo)
;Value: (define ((compose f g) x) (f (g x)))
```

```
;;; --------------------------
;;; Other Compound Types
;;; --------------------------

;;; There are other useful compound data types:
;;; Vectors, strings, characters, hash tables,
;;; records (see documentation)

(define v (vector 10 20 30))

(vector-ref v 2)
;Value: 30

(string-length "hello")
;Value: 5
```

```
;;; --------------------------
;;; Mutation / Assignment
;;; --------------------------

;;; You can also assign to variables and mutate
;;; compound data structure.  Procedures with
;;; mutations generally are named ending with a
;;; "!"  Be careful / generally use only when
;;; needed

(set! pi 3.14)
;Value: 3.141592653589793

(define p (cons 1 2))

p
;Value 16: (1 . 2)

(set-car! p 3)
;Unspecified return value

p
;Value 16: (3 . 2)
```