SICP Notes

Sooheon Kim

Summer 2015

Contents

1 Building Abstractions with Procedures

Definition of square root: $\sqrt{}(x)$ = the y such that $y >= 0$ and y^2 = x

This definition is a valid mathematical function which describes the relationship between a number and its root. It is *not* a valid process by which a square root may be derived. A function is declarative knowledge (the properties of a thing), and a procedure is *imperative* knowledge (how to do a thing).

1.1 Heron of Alexandria's Square Root Algorithm

To find sqrt(x) approximately:

1. Make a guess G
2. Improve guess by averaging G and X
3. Keep improving G until it is good enough
4. Use 1 as an initial guess

Guess	Quotient	Avg.
1	2/1 = 2	((2+1)/2) = 1.5
1.5	2/1.5 = 1.3333	((1.3333+1.5)/2) = 1.4167
1.4167	2/1.4167 = 1.4118	((1.4167+1.4118)/2) = 1.4142
1.4142	...	...

1.1.1 Naive Version

```
(defn abs [n]
  (if (< n 0)
    (- n)
    n))

(defn square [n]
  (* n n))

(defn average [x y]
  (/ (+ x y) 2))

(defn improve [guess x]
  (average guess (/ x guess)))

(defn good-enough? [guess x]
  (< (abs (- (square guess) x)) 0.001))

(defn sqrt-iter [guess x]
  (if (good-enough? guess x)
    guess
    (sqrt-iter (improve guess x) x)))

(defn sqrt [x]
  (sqrt-iter 1 x))

(sqrt 2.0)
;; => 1.4142156862745097
```

1.1.2 Blocked Version

Keeping helper functions `good-enough?` and others from polluting namespace. This idea originated with Algol 60.

```
(defn sqrt2 [x]
  (defn good-enough? [guess x]
    (< (abs (- (square guess) x)) 0.001))
  (defn improve [guess x]
    (/ (+ guess (/ x guess)) 2))
  (defn sqrt-iter [guess x]
```

```
    (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))
  (sqrt-iter 1.0 x))

(sqrt2 2)
;; => 1.4142156862745097
```

1.1.3 Exercise 1.6

Define `if` in terms of `cond`:

```
(defn new-if [predicate then-clause else-clause]
  (cond
    predicate then-clause
    :else else-clause))

(new-if (= 2 3) 0 5)
;; => 5
```

Use `new-if` to write `square-root`:

```
(defn new-sqrt-iter [guess x]
  (new-if (good-enough? guess x)
          guess
          (new-sqrt-iter (improve guess x)
                         x)))
```

What happens when we eval this?

```
(new-sqrt-iter 1 2)

(new-if (good-enough? 1 2)
        1
        (new-sqrt-iter (improve 1 2) 2))

(new-if false
        1
        (new-sqrt-iter (improve 1 2) 2))

(new-sqrt-iter (improve 1 2) 2)
```

...and so on, ad infinitum. This will result in infinite recursion, due to applicative-order evaluation. Because both the `then-clause` and `else-clause` operands are evaluated before being applied to the `cond` operator. Even when the predicate is true, the `else-clause` is evaluated, keeping the procedure from ever resolving.

1.1.4 Exercise 1.7

`good-enough?` will be inaccurate for very small numbers. It is also inadequate for very large numbers. Why?

```
(sqrt 0.0001)
;; => 0.03230844833048122
;; Should be 0.01

(defn really-good-enough? [guess oldguess]
  "Stop if change after iteration is less than 0.01%"
  (<= (abs (- guess oldguess))
      (abs (* guess 0.0001))))

(defn sqrt3-iter [guess oldguess x]
  (if (really-good-enough? guess oldguess)
    guess
    (sqrt3-iter (improve guess x) guess x)))

(defn sqrt3 [x]
  (sqrt3-iter 1.0 2 x))

(sqrt3 0.0001)
;; => 0.010000000025490743
```

1.1.5 Exercise 1.8

Newton's method for cube roots is based on the fact that if y is an approximation to the cube root of x, then a better approximation is given by the value:

$$\frac{x/y^2+2y}{3}$$

```
(defn cbrt-good-enough? [guess oldguess]
  "Stop if change after iteration is less than 0.01%"
  (<= (abs (- guess oldguess))
```

```
      (abs (* guess 0.0001))))

(defn cbrt-improve [guess x]
  (/ (+ (/ x (* guess guess)) guess guess)
     3))

(defn cbrt-iter [guess oldguess x]
  (if (cbrt-good-enough? guess oldguess)
    guess
    (cbrt-iter (cbrt-improve guess x) guess x)))

(defn cbrt [x]
  (cbrt-iter 1.0 0 x))

(cbrt 42)
;; => 3.476026657071078
```

1.2 Procedures as Black-Box Abstractions

Defining procedures is a way to suppress detail. By forming black boxes around procedures which behave in expected ways, we no longer have to consider the details of the inner mechanism.

Parameters passed to black-box procedures are local to that procedure.

1.2.1 Lexical Scoping

We can have a `naive sqrt procedure`, leaves all its helper processes lying about. A smarter `blocked version` will put the helper procedures within the top level sqrt procedure. However, because they all share a lexical scope, and the variable x is unchanging within this scope, it is unnecessary to explicitly pass x to each procedure.

```
(defn lexical-sqrt [x]
  (defn good-enough? [guess]
          (< (abs (- (square guess) x)) 0.001))
  (defn improve [guess]
    (/ (+ guess (/ x guess)) 2))
  (defn iter [guess]
    (if (good-enough? guess)
      guess
      (iter (improve guess))))
```

```
  (iter 1.0))

(lexical-sqrt 2)
;; => 1.4142156862745097
```

2 Procedures and the Processes They Generate

2.1 Substitution Rule

To evaluate an application:

1. Evaluate the operator to get procedure
2. Evaluate the operands to get arguments
3. Apply the procedure to the arguments
   (a) Copy the body of the procedure, substituting the arguments supplied for the formal parameters of the procedure
   (b) Evaluate the resulting new body

2.2 Linear Recursion and Iteration

```
(defn recursive-factorial [n]
  (if (= n 1)
    1
    (* n (recursive-factorial (dec n)))))

(factorial 5)
(* 5 (factorial 4))
(* 5 (* 4 (factorial 3)))
(* 5 (* 4 (3 (factorial 2))))
(* 5 (* 4 (* 3 (* 2 (factorial 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
;; => 120

(defn fact-iter [product count max-count]
```

```
  (if (> count max-count)
    product
    (fact-iter (* count product)
               (inc count)
               max-count)))

(defn iterative-factorial [n]
  (fact-iter 1 1 n))

(iterative-factorial 5)
(fact-iter 1 1 5)
(fact-iter 1 2 5)
(fact-iter 2 3 5)
(fact-iter 6 4 5)
(fact-iter 24 5 5)
(fact-iter 120 6 5)
;; => 120
```

An iteration has all of its state in explicit variables. If the program is killed and restarted, it can continue. Recursion keeps some information "under the table", outside of explicit variables given to the program.

```
(defn fib [n]
  (if (< n 2)
    n
    (+ (fib (- n 1))
       (fib (- n 2)))))
```

2.2.1 Recursive Process vs. Procedure

A recursive procedure is one whose definition refers (directory or indirectly) to itself. A linearly recursive process, on the other hand, is recursive in the way its evaluation evolves not how it is written. `fact-iter` is a recursive *procedure* which generates an iterative *process*. Its state is described completely by its three variables, and the interpreter need only keep track of them to execute the process.

In many languages (Ada, Pascal, C...), recursive procedures take memory growing with the number of procedure calls, even when the *process* is iterative. In these languages, iterative processes can only be defined with special-purpose "looping constructs": `do`, `repeat`, `until`, `for`, etc. Scheme

can execute an iterative process (described by a recursive procedure) in constant space. This is called *tail-recursion*.

2.2.2 Exercise 1.9

Both processes below define a method for adding two positive integers. Using the substitution model, illustrate the process generated by each procedure in evaluating `(+ 4 5)`. Are these processes iterative or recursive?

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))

(+ 4 5)
(inc (+ (dec 4) 5))
(inc (+ 3 5))
(inc (inc (+ 2 5)))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
;; => 9
```

```
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))

(+ 4 5)
(+ (dec 4) (inc 5))
(+ 3 6)
(+ 2 7)
(+ 1 8)
(+ 0 9)
;; => 9
```

The first is a recursive process, the second iterative. They are both recursively defined procedures.

2.2.3 Exercise 1.10

```
(defn ack [x y]
  (cond
    (= y 0) 0
    (= x 0) (* 2 y)
    (= y 1) 2
    :else (ack (dec x)
               (ack x (dec y)))))

(ack 1 10)
(ack 0 (ack 1 9))
(ack 0 (ack 0 (ack 1 8)))
(ack 0 (ack 0 (ack 0 (ack 1 7))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 1 6)))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 5))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 4)))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 3))))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 2)))))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 1))))))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 2)))))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 4)))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 8)))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 16))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 32)))))
(ack 0 (ack 0 (ack 0 (ack 0 64))))
(ack 0 (ack 0 (ack 0 128)))
(ack 0 (ack 0 256))
(ack 0 512)
;; => 1024

(ack 2 4)
(ack 1 (ack 2 3))
(ack 1 (ack 1 (ack 2 2)))
(ack 1 (ack 1 (ack 1 (ack 2 1))))
(ack 1 (ack 1 (ack 1 2)))
(ack 1 (ack 1 (ack 0 (ack 1 1))))
(ack 1 (ack 1 (ack 0 2)))
(ack 1 (ack 1 4))
(ack 1 (ack 0 (ack 1 3)))
```

```
(ack 1 (ack 0 (ack 0 (ack 1 2))))
(ack 1 (ack 0 (ack 0 (ack 0 (ack 1 1)))))
(ack 1 (ack 0 (ack 0 (ack 0 2))))
(ack 1 (ack 0 (ack 0 4)))
(ack 1 (ack 0 8))
(ack 1 16)
(ack 0 (ack 1 15))
(ack 0 (ack 0 (ack 1 14)))
(ack 0 (ack 0 (ack 0 (ack 1 13))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 1 12)))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 11))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 1 10)))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 (ack 0 1024))))))
(ack 0 (ack 0 (ack 0 (ack 0 (ack 0 2048)))))
(ack 0 (ack 0 (ack 0 (ack 0 4096))))
(ack 0 (ack 0 (ack 0 8192)))
(ack 0 (ack 0 16384))
(ack 0 32768)
;; => 65536

(ack 3 3)
(ack 2 (ack 3 2))
(ack 2 (ack 2 (ack 3 1)))
(ack 2 (ack 2 2))
(ack 2 (ack 1 (ack 2 1)))
(ack 2 (ack 1 2))
(ack 2 (ack 0 (ack 1 1)))
(ack 2 (ack 0 2))
(ack 2 4)
;; => 65536

(defn f [n]
  (ack 0 n))
(f 100)
;; => 200

;; f(x) = 2x

(defn g [n]
  (ack 1 n))
```

```
(= (g 50)
   (apply * (repeat 50 2N)))
;; => true

;; g(x) = 2^x

(defn h [n]
  (ack 2 n))

;; h(n) = 2^2^... (n times)
```

2.3 Tree Recursion

```
(defn fib [n]
  (cond
    (= n 0) 0
    (= n 1) 1
    :else (+ (fib (dec n))
             (fib (dec (dec n))))))
```

Because `fib` calls itself twice each time it is invoked, the process ends up looking like a tree. The number of steps required in such a process is proportional to the number of nodes in the tree, and the space required is proportional to the maximum depth.

```
(defn fib-iter [a b count]
  (if (= count 0)
    b
    (fib-iter (+ a b) a (dec count))))

(defn fib [n]
  (fib-iter 1 0 n))

(map fib (range 1 11))
;; => (1 1 2 3 5 8 13 21 34 55)
```

2.3.1 Counting change

How many different ways can we make change of $1.00, given half-dollars, quarters, dimes, nickels, and pennies? More generally, can we write a procedure to compute the number of ways to change any given amount of money?

The number of ways to change amount a using n kinds of coins equals

- The number of ways to change amount a using all but the first kind of coin, plus
- The number of ways to change amount a - d using all n kinds of coins, where d is the denomination of the first kind of coin.

Keeping in mind that

- If a is exactly 0, we should count that as 1 way to make change.
- If a is less than 0, we should count that as 0 ways to make change.
- If n is 0, we should count that as 0 ways to make change.

```
(defn first-denomination [kinds-of-coins]
  (cond
    (= kinds-of-coins 1) 1
    (= kinds-of-coins 2) 5
    (= kinds-of-coins 3) 10
    (= kinds-of-coins 4) 25
    (= kinds-of-coins 5) 50))

(defn cc [amount kinds-of-coins]
  (cond
    (= amount 0) 1
    (or (< amount 0) (= kinds-of-coins 0)) 0
    :else (+ (cc amount (dec kinds-of-coins))
             (cc (- amount
                    (first-denomination kinds-of-coins))
                 kinds-of-coins))))

(defn count-change [amount]
  (cc amount 5))

(count-change 11)
(cc 11 5)
(cond (= 11 0) 1 (or (< 11 0) (= 5 0)) 0 :else (+ (cc 11 4) (cc (- 11 50) 5)))
(+ (cc 11 4) (cc -39 5))
(+ (+ (cc 11 3) (cc (- 11 25) 4)) 0)
(+ (cc 11 3) 0)
;; => 4
```

2.3.2 Exercise 1.11

A function f is defined by the rule that f(n) = n if n<3 and f(n) = f(n - 1) + 2f(n - 2) + 3f(n - 3) if n> 3. Write a procedure that computes f by means of a recursive process. Write a procedure that computes f by means of an iterative process.

```
(defn f
  "Version with recursive process"
  [n]
  (if (< n 3)
    n
    (+ (f (dec n))
       (* 2 (f (- n 2)))
       (* 3 (f (- n 3))))))

(map #(vector % (f %)) (range 1 10))
;; =>
;; ([1 1]
;;  [2 2]
;;  [3 4]
;;  [4 11]
;;  [5 25]
;;  [6 59]
;;  [7 142]
;;  [8 335]
;;  [9 796])

(f 3)
(+ (f 2)
   (* 2 (f 1))
   (* 3 (f 0)))
;; => 4

(f 4)
(+ (+ (f 2)
      (* 2 (f 1))
      (* 3 (f 0)))
   (* 2 (f 2))
   (* 3 (f 1)))
;; => 11
```

```
(defn f-iter [a b c count]
  (cond
    (< count 3) count
    (= count 3) (+ c (* 2 b) (* 3 a))
    :else (f-iter b c (+ c (* 2 b) (* 3 a)) (dec count))))

(defn iterative-f [n]
  (f-iter 0 1 2 n))

;; Two functions give the same results
(for [n (range 1 10)]
  (= (iterative-f n) (f n)))
;; =>
;; (true
;;  true
;;  true
;;  true
;;  true
;;  true
;;  true
;;  true
;;  true)
```

2.3.3 Exercise 1.12

Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

```
(defn pascal-layer
  "Take a non-negative number n and returns the nth slice of Pascal's triangle"
  [n]
  (let [gen-next (fn [prev]
                   (map (partial apply +) (partition 2 1 prev)))]
    (cond
      (= 0 n) '(1)
      (= 1 n) '(1 1)
      :else (concat [1] (gen-next (pascal-layer (dec n))) [1]))))

(map pascal-layer (range 9))
```

```
;; =>
;; ((1)
;;  (1 1)
;;  (1 2 1)
;;  (1 3 3 1)
;;  (1 4 6 4 1)
;;  (1 5 10 10 5 1)
;;  (1 6 15 20 15 6 1)
;;  (1 7 21 35 35 21 7 1)
;;  (1 8 28 56 70 56 28 8 1))

(defn nextrow [row]
  (vec (concat [1] (map #(apply + %) (partition 2 1 row)) [1] )))

(def pascal
  (iterate #(concat [1]
                    (map + % (rest %))
                    [1])
           [1]))
```

2.3.4 Tower of Hanoi

```
(defn move [n from to spare]
  (cond (= 0 n) "done"
        :else (do (move (dec n) from spare to)
                  (move (dec n) spare to from))))
```

2.4 Exponentiation

Linearly recursive definition, requires O(n) steps and O(n) space.

```
(defn expt [b n]
  (if (= n 0)
    1
    (* b (expt b (dec n)))))

(expt 2 5)
(* 2 (expt 2 4))
(* 2 (* 2 (expt 2 3)))
(* 2 (* 2 (* 2 (expt 2 2))))
(* 2 (* 2 (* 2 (* 2 (expt 2 1)))))
```

```
(* 2 (* 2 (* 2 (* 2 (* 2 (expt 2 0))))))
(* 2 (* 2 (* 2 (* 2 (* 2 1)))))
(* 2 (* 2 (* 2 (* 2 2))))
(* 2 (* 2 (* 2 4)))
(* 2 (* 2 8))
(* 2 16)
;; => 32
```

Faster version, taking advantage of the fact that $b^n = (b^{(n/2)})^2$ if n is even, and $b^n = b * b^{(n-1)}$ if odd. This grows in O(log(n)) time.

```
(defn square [n]
  (* n n))

(defn fast-expt [b n]
  (cond (= n 0) 1
        (even? n) (square (fast-expt b (/ n 2)))
        :else (* b (fast-expt b (dec n)))))

(fast-expt 2 6)
(square (fast-expt 2 3))
(square (* 2 (fast-expt 2 2)))
(square (* 2 (square (fast-expt 2 1))))
(square (* 2 (square (* 2 (fast-expt 2 0)))))
(square (* 2 (square (* 2 1))))
(square (* 2 (square 2)))
(square (* 2 4))
(square 8)
;; => 64
```

O(log(n)) and O(n) are hugely different as n grows large.

```
(time (expt 2N 100))

(time (fast-expt 2N 50))
```

2.4.1 Exercise 1.16

Design a procedure that evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps, as does fast-expt. (Hint: Using the observation that (bn/2)2 = (b2)n/2, keep, along with the

exponent n and the base b, an additional state variable a, and define the state transformation in such a way that the product a bn is unchanged from state to state. At the beginning of the process a is taken to be 1, and the answer is given by the value of a at the end of the process. In general, the technique of defining an invariant quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

```
(defn iterative-expt [b n]
  (let [iter (fn [a b n]
               (cond
                 (= 0 n) a
                 (even? n) (iter a (square b) (/ n 2))
                 :else (iter (* a b) b (dec n))))])
  (iter 1 b n))

(iterative-fast-expt 2 10)

;; => 1024
```

2.4.2 Exercise 1.17

The exponentiation algorithms in this section are based on performing exponentiation by means of repeated multiplication. In a similar way, one can perform integer multiplication by means of repeated addition. The following multiplication procedure (in which it is assumed that our language can only add, not multiply) is analogous to the expt procedure:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

This algorithm takes a number of steps that is linear in b. Now suppose we include, together with addition, operations double, which doubles an integer, and halve, which divides an (even) integer by 2. Using these, design a multiplication procedure analogous to fast-expt that uses a logarithmic number of steps.

```
(defn double [n]
  (* n 2))
```

```
(defn halve [n]
  (/ n 2))

(defn multiply [a b]
  (cond
    (= 0 b) 0
    (even? b) (multiply (double a) (halve b))
    :else (+ a (multiply a (dec b)))))

(multiply 7 100)
;; => 700
```

2.4.3 Exercise 1.18

Using the results of exercises 1.16 and 1.17, devise a procedure that generates an iterative process for multiplying two integers in terms of adding, doubling, and halving and uses a logarithmic number of steps.

```
(defn *-iter [acc a b]
  (cond
    (= b 0) acc
    (even? b) (*-iter acc (double a) (halve b))
    :else (*-iter (+ acc a) a (dec b))))

(defn iterative-* [a b]
  (*-iter 0 a b))

(iterative-* 8 9)
;; => 72
```

2.4.4 Exercise 1.19

```
(defn fib-iter [a b p q count]
  (cond
    (= count 0) b
    (even? count) (fib-iter a
                            b
                            (+ (* p p) (* q q))
                            (+ (* 2 p q) (* q q))
                            (/ count 2))
    :else (fib-iter (+ (* b q) (* a q) (* a p))
```

```
                    (+ (* b p) (* a q))
                    p
                    q
                    (dec count))))
(defn fib [n]
  (fib-iter 1 0 0 1 n))

(map fib (range 1 11))
;; =>
;; (1 1 2 3 5 8 13 21 34 55)
```

2.5 Greatest Common Denominators

The idea of the algorithm is based on the observation that, if r is the remainder when a is divided by b, then the common divisors of a and b are precisely the same as the common divisors of b and r. Thus, we can use the equation

GCD(a,b) = GCD(b,r)

To successively reduce the problem of computing a GCD to the problem of computing the GCD of smaller and smaller pairs of integers. For example,

GCD(206,40) = GCD(40,6) = GCD(6,4) = GCD(4,2) = GCD(2,0) = 2

```
(defn gcd
  "An iterative process in log(n) time"
  [a b]
  (if (= b 0)
    a
    (gcd b (rem a b))))

(gcd 206 40)
;; => 2
```

This is known as *Euclid's Algorithm.*

2.5.1 Lamé's Theorem

If Euclid's Algorithm requires k steps to compute the GCD of some pair, then the smaller number in the pair must be greater than or equal to the kth Fibonacci number.

2.6 Testing for Primality

2.6.1 Searching for divisors

The following program finds the smallest integral divisor (greater than 1) of a given number n. It does this in a straightforward way, by testing n for divisibility by successive integers starting with 2.

```
(defn divides? [a b]
  (= (rem b a) 0))

(defn find-divisor [n test-divisor]
  (cond
    (> (square test-divisor) n) n
    (divides? test-divisor n) test-divisor
    :else (find-divisor n (inc test-divisor))))

(defn smallest-divisor [n]
  (find-divisor n 2))

(smallest-divisor 25)
;; => 5

(defn prime? [n]
  (= n (smallest-divisor n)))

(prime? 25)
;; => false

(prime? 11)
;; => true
```

The algorithm tests divisors between 1 and sqrt(n). Therefore, the number of steps required to identify n as prime will have an order of growth O(sqrt(n))

2.6.2 The Fermat test

Fermat's Little Theorem: If n is a prime number and a is any positive integer less than n, then a raised to the nth power is congruent to a modulo n.

(Two numbers are said to be congruent modulo n if they both have the same remainder when divided by n. The remainder of a number a when divided by n is also referred to as the remainder of a modulo n, or simply as a modulo n.)

```
(defn expmod [base exp m]
  (cond
    (= exp 0) 1
    (even? exp) (rem (square (expmod base (/ exp 2) m))
                     m)
    :else (rem (* base (expmod base (dec exp) m))
               m)))

(defn fermat-test [n]
  (defn try-it [a]
    (= (expmod a n n) a))
  (try-it (+ 1 (rand-int (dec n)))))

(defn fast-prime? [n times]
  (cond
    (= times 0) true
    (fermat-test n) (fast-prime? n (dec times))
    :else false))

(fast-prime? 31939 100)
;; => true
```

3 Higher Order Procedures

```
(defn sum-int
  "Sums integers from a to b"
  [a b]
  (if (> a b)
    0
    (+ a (sum-int (inc a) b))))

(sum-int 1 10)
;; => 55

(defn sum-of-squares
```

```
  "Sums the squares of the integers from a to b"
  [a b]
  (if (> a b)
    0
    (+ (square a) (sum-of-squares (inc a) b))))

(sum-of-squares 1 10)
;; => 385

(defn sigma
  "Procedure for generating summation procedures. Term denotes the lower bound
  index, and next gives the next index."
  [term a next b]
  (if (> a b)
    0
    (+ (term a)
       (sigma term (next a) next b))))

(defn sum-int-2 [a b]
  (sigma identity a inc b))

(sum-int-2 1 10)
;; => 55

(defn sum-of-squares-2 [a b]
  (sigma square a inc b))

(sum-of-squares-2 1 10)
;; => 385

(defn pi-sum-2 [a b]
  (sigma (fn [i] (/ 1 (* i (+ i 2))))
         a
         (fn [i] (+ i 4))
         b))
(pi-sum-2 1.0 100)
;; => 0.3901993315738763
```

3.1 Improving Heron of Alexandria's Square Root Algorithm

`This` algorithm for computing square roots is effective but difficult to parse. What it is essentially trying to do is find the fixed point of some function which gives the square root.

```
(defn fixed-point
  "Returns the fixed point of the function f starting with start, iterating the
  function until the results converge within a given tolerance."
  [f start]
  (let [tolerance 0.00001
        close-enough? (fn [u v] (< (abs (- u v)) tolerance))]
    (loop [old start
           new (f start)]
      (if (close-enough? old new)
        new
        (recur new (f new))))))

  (defn sqrt
    "The square root of x is the fixed point of that procedure which takes y and
    averages x/y with y."
    [x]
    (fixed-point (fn [y] (average (/ x y) y))
                 1))

  (sqrt 2.0)
  ;; => 1.4142135623746899
```

But why the need to find the fixed point of f(y) = (y + x/y)/2 ? This is counter intuitive. More directly obvious is that the fixed point of f(y) = x/y will give a value y which is the square root of x. But procedure can't be used directly. If 1 is given as a starting value y and x is 2, x/y is 2. 2/2 is 1. This is an oscillation, which will never converge. The way you get an oscillation to converge is by averaging it. Expressing this abstraction:

```
(defn avg-damp
  "Averages the last value with the value returned by running f on it."
  [f]
  (fn [x] (average (f x) x)))
```

```
(defn sqrt
  "Square root of x is the fixed point of the procedure resulting from
  average-damp."
  [x]
  (fixed-point (avg-damp (fn [y] (/ x y)))
               1))

(sqrt 2.0)
;; => 1.4142135623746899
```

3.2 Newton's Method

A method for finding the roots of a function.

To find a y such that f(y) = 0: Start with a guess, y_0; $y_n+1 = y_n$ - $f(y_n)/(df/dy|y = y_n)$

```
(def dx 0.000001)

(defn deriv [f]
  (fn [x] (/ (- (f (+ x dx))
                (f x))
             dx)))

(defn newton [f guess]
  (let [df (deriv f)]
    (fixed-point (fn [x] (- x (/ (f x) (df x))))
                 guess)))

(defn sqrt
  "The value of y for which (- x (* y y)) => 0 is the square root of x."
  [x]
  (newton (fn [y] (- x (* y y)))
          1))

(sqrt 2.0)
;; => 1.4142135623754424
```

3.3 Rights and Privileges of First-class Citizens (in a programming language)

1. To be named by variables

2. To be passed as arguments to procedures
3. To be returned as values of procedures
4. To be incorporated into data structures

4 Compound Data

In writing square root, we contract out some work to good-enough?. We don't care exactly what it does, as long as it takes and returns the right values. We divorce the task of building things from the task of implementing the parts. Isolating details!

4.1 Rational Numbers

By wishful thinking.

```
(defn make-rat
  "Take a numerator and denominator and returns a 'cloud' which represents a
  rational number"
  [n d]
  {:numer n :denom d})

;; (make-rat n d) => cloud
;; (:numer cloud) => n
;; (:denom cloud) => d

(defn +rat [x y]
  (make-rat
   (+ (* (:numer x) (:denom y))
      (* (:numer y) (:denom x)))
   (* (:denom x) (:denom y))))

(defn *rat [x y]
  (make-rat
   (* (:numer x) (:numer y))
   (* (:denom x) (:denom y))))
```