1.3 - Formulating Abstractions with Higher-Order Procedures

- Procedures/definitions in powerful programming languages allow for abstraction
- *Higher-order procedures*
    - Able to accept procedures as arguments and able to return procedures as values

1.3.1 - Linear Recursion and Iteration

- e.g.

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))

(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))

(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

- All have a common underlying pattern

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b))))
```

- e.g.

```
(define (sum term a b next)
  (if (> a b)
    0
     (+ (term a)
       (sum term (next a) b next))))

(define (inc n)
  (+ n 1))
```

```
(define (cube n)
  (* n n n))

(define (sum-of-cubes a b)
  (sum cube a b inc))

(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))))
  (define (pi-next x)
    (+ x 4))
  (sum pi-term a b pi-next))
```

Exercise 1.29

- No idea, but searched up

```
 (define (round-to-next-even x)
   (+ x (remainder x 2)))
 (define (simpson f a b n)
   (define fixed-n (round-to-next-even n))
   (define h (/ (- b a) fixed-n))
   (define (simpson-term k)
     (define y (f (+ a (* k h))))
     (if (or (= k 0) (= k fixed-n))
         (* 1 y)
         (if (even? k)
             (* 2 y)
             (* 4 y))))
   (* (/ h 3) (sum simpson-term 0 inc fixed-n)))

(define (sum term a next b)
  (if (> a b)
      0
      (sum term (next a) next b)))
```

Exercise 1.30

```
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ (term a) result))))
  (iter a 0))
```

Exercise 1.31

```
;basic defs
(define (square x)
  (* x x))

(define (even? x)
  (= (remainder x 2) 0))

;recursive product procedure
(define (product a term b next)
  (if (> a b)
      1
      (* (term a)
         (product (next a) term b next))))

;iterative product procedure
(define (product-iter a term b next)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* (term a) result))))
  (iter a 1))

;prodcut-iter test
(define (fact-iter n)
  (define (inc x)
    (+ x 1))
  (define (fact-iter-pre a b)
    (product-iter a inc b inc))
  (fact-iter-pre 1 (- n 1)))

;recursive factorial w/ product procedure
(define (pre-factorials a b)
  (define (inc x)
    (+ x 1))
  (product a inc b inc))

(define (factorials n)
  (pre-factorials 0 (- n 1)))

;approximating pi/4 recursively w/ product procedure
(define (pre-pi-4 a b)
```

```
  (define (inc z)
    (+ z 1))
  (define (pi-4-term y)
    (if (even? y)
        (/ (+ y 2) (+ y 1.0))
          (/ (+ y 1.0) (+ y 2))))
  (product-iter a pi-4-term b inc))

(define (estimate-pi accuracy)
  (* (pre-pi-4 1 accuracy) 4))
```

Exercise 1.32

```
;accumulate high order procedure
(define (accumulate combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
                (accumulate combiner
                            null-value
                            term
                           (next a)
                            next
                            b))))

;accumulate as iterative procedure
(define (accumulate-iter combiner null-value term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner (term a)
                                 result))))
  (iter a null-value))

;sum in terms of accumulate
(define (sum term a next b)
  (define (combiner x y)
    (+ x y))
   (accumulate-iter combiner 0 term a next b))

;product in terms of accumulate
(define (product term a next b)
```

```
  (define (combiner x y)
    (* x y))
  (accumulate-iter combiner 1 term a next b))

;trial for sum procedure
(define (range-sum a b)
  (define (inc x)
    (+ x 1))
  (sum inc (- a 1) inc (- b 1)))
```

Exercise 1.33

```
;definitions
(define (square x)
  (* x x))

;prime
(define (rab-mill-expmod base exp m)
  (define (check-sqr pre-sqr)
    (define (check pre-sqr sqr)
      (if (and (not (= pre-sqr 1))
               (not (= pre-sqr (- m 1)))
               (= sqr 1))
          0
          sqr))
    (check pre-sqr (remainder (square pre-sqr) m)))
  (cond ((= exp 0) 1)
        ((even? exp)
         (check-sqr (rab-mill-expmod base (/ exp 2) m)))
        (else (remainder (* base (rab-mill-expmod base
                                                   (- exp 1)
                                                   m))  m))))

(define (rab-mill-recur n)
  (define (try a)
    (define (check term)
      (and (not (= term 0))
           (= term 1)))
    (check (rab-mill-expmod a (- n 1) n)))
  (try (if (> n 4294967087)
           (+ 1 (random 4294967087))
           (+ 1 (random (- n 1))))))
```

```
(define (prime? n)
  (define (prime-iter n times)
    (cond ((< n 2) false)
          ((= times 0) true)
          ((rab-mill-recur n) (prime-iter n (- times 1)))
          (else false)))
  (prime-iter n 100))


;accumulate filter
(define (accumulate-filter filter combiner null-v term a next b)
  (if (> a b)
      null-v
      (if (filter a)
      (combiner (term a)
                (accumulate-filter filter
                                   combiner
                                   null-v
                                   term
                                   (next a)
                                   next b))
      (combiner null-v
                (accumulate-filter filter
                                   combiner
                                   null-v
                                   term
                                   (next a)
                                   next b)))))


;sum of squares of primes in a range
(define (sum-prime-sqrs a b)
  (define (filter x)
    (prime? x))
  (define (combiner y z)
    (+ y z))
  (define (term x)
    (* x x))
  (define (next x)
    (+ x 1))
  (accumulate-filter filter combiner 0 term a next b))
```

```
;product positve integers less than n relative prime to n
(define (product-relative-prime n)
  (define (filter a)
    (= 1 (gcd n a)))
  (define (combiner x y)
    (* x y))
  (define (inc x)
    (+ x 1))
  (define (identity x)
    (* x 1))
  (accumulate-filter filter combiner 1 identity 1 inc n))
```

1.3.2 - Constructing Procedures Using Lambda

- Special form
  - *Lamba*
    (lambda <formal parameters> <body>)
    - Does Not associate with a name in the environment
    - Yet, works same as a definition
    - Can be used as operators
      - i.e. ((lambda (x y z) (+ x y (square z))) 1 2 3)
  - *Let*
    -

e.g.
(define (plus4 x) (+ x 4)) == (define (plus4 (lambda (x) (+ x 4)))