Number theoretic functions

```
(define (add-single-carry x y c)
 (if (and (is-single x) (is-single y) (is-single c)) all operators
have a multi-argument version
     (+ x y c)
     (error "Operands should be single digit")))

(define (addc x y c)
  (cond [(and (= x 0) (= y 0)) c]
        [#t (let* [(qx10 (quotient x 10))
                   (rx10 (remainder x 10))
                    (qy10 (quotient y 10))
                   (ry10 (remainder y 10))
                   (sumc (add-single-carry rx10 ry10 c))
                   (qsum10 (quotient sumc 10))
                   (rsum10 (remainder sumc 10))]
              (convert (addc qx10 qy10 qsum10) rsum10))]))
(define (convert x y) (+ (* x 10) y))

(define (is-single x) (= (quotient x 10) 0))

(define (add x y) (addc x y 0))




(define (mult x y)
 (if (= y 0) 0
     (let [(qy10 (quotient y 10))
           (ry10 (remainder y 10))]
       (add (* (mult x qy10) 10) (multn1c x ry10 0)))))

(define (multn1c x y c)    y is single digit, c is carry
 (if (= x 0) c
     (let* [(qx10 (quotient x 10))
            (rx10 (remainder x 10))
            (prodc (add (mult-single rx10 y) c))
            (qp10 (quotient prodc 10))
            (rp10 (remainder prodc 10))]
       (convert (multn1c qx10 y qp10) rp10))))
```

```
(define (mult-single x y)
 (if (and (is-single x) (is-single y)) all operators have a
multi-argument version
     (* x y)
     (error "Operands should be single digit")))
```

The rest of the number theoretic functions appear as solutions of the minor assignments.

Introduction to higher order functions

```
(define (tan x k) (tan-helper x k 1))

(define (tan-helper x k c)
 (cond [(> c k) (* x x)]
       [else (if (= c 1)
                 (/ x (- (- (* 2 c) 1)
                         (tan-helper x k (+ c 1))))
                 (/ (* x x) (- (- (* 2 c) 1)
                              (tan-helper x k (+ c 1)))))]))


(define (sin x k) (sin-helper x k 1))

(define (sin-helper x k c)
 (cond [(> c k) (/ x (fac (- (* 2 c) 1)))]
       [else (- (/ x (fac (- (* 2 c) 1)))
                (* (* x x) (sin-helper x k (+ c 1))))]))

(define (fac n)
 (if (= n 0) 1
    (* n (fac (- n 1)))))

(define (hof f g n m x k)
 (define (hof-helper c)
   (if (> c k) (n c)
       (f (n c)
          (g (m c)
             (hof-helper (+ c 1))))))
(hof-helper 1))
```

Define tan sin etc using hof...

```
(define (tan x k)
 (define (n i) (if (= i 1) x (* x x)))
 (define (m i) (- (* 2 i) 1))
 (hof / - n m x k))

(define (sin x k)
 (define (n i) (/ x (fac (- (* 2 i) 1))))
 (define (m i) (expt x 2))
 (hof - * n m x k))

(define (chained-sqrt x k)
  (define (f x y) (sqrt y))
  (define (n i) 0)
  (define (m i) x)
  (hof f + n m x k))

(define (balance pr r y mp)
 (if (= y 0) pr
     (balance (- (+ pr (* pr (/ r 100))) (* mp 12)) r (- y 1) mp)))

(define (until p f x) (if (p x) x (until p f (f x))))

(define (diff f)
 (define delta 0.0001)
 (lambda (x) (/ (- (f (+ x delta)) (f x)) delta)))

(define (zero f)(define initial 1.0)
 (define (improve xi)
   (- xi (/ (f xi) ((diff f) xi))))
 (define (close-enough x) (< (abs (f x)) 0.00001))
 (until close-enough improve initial))

(define (emi pr r y)
 (zero (lambda (mp) (balance pr r y mp))))
```

Introduction to lists

```
(define (sum l)
 (if (null? l) 0
     (+ (car l) (sum (cdr l)))))

(define (product l)
 (if (null? l) 1
     (+ (car l) (product (cdr l)))))

(define (reverse l)
 (if (null? l) `()
     (append (reverse (cdr l)) (list (car l)))))

(define (append l1 l2)
   (if (null? l1) l2
     (cons (car l1) (append (cdr l1) l2))))

(define (map f l)
 (if (null? l) `()
     (cons (f (car l)) (map f (cdr l)))))

(define (filter p l)
 (cond [(null? l) `()]
       [(p (car l))(cons (car l) (filter p (cdr l)))]
       [else (filter p (cdr l))]))
+

(define (foldr op id l)
 (if (null? l) id
     (op (car l) (foldr op id (cdr l)))))

(define (sum l) (foldr + 0 l))

(define (product l) (foldr * 1 l))

(define (reverse l)
 (define (op x y) (append y (list x)))
 (foldr op `() l))
```

```
(define (append l1 l2)
 (define op cons)
 (foldr cons  l2 l1))

(define (map f l)
 (define (op x y) (cons (f x) y))
(foldr op `() l))

(define (filter p l)
 (define (op x y)
   (if (p x) (cons x y) y))
 (foldr op `() l))

(define (inits l)
 (if (null? l) (cons `() `())
     (cons '() (map (lambda (l1) (cons (car l) l1)) (inits (cdr
l))))))

(define (tails l)
 (if (null? l) `(())
     (cons l (tails (cdr l)))))

(define (perms l)
 (define (g x y)
   (append (map (lambda (l1) (cons x l1))
                (perms (remove x l)))
           y))
  (cond ((null? l) '(()))
        (else (foldr g '() l))))

(define (choose n l)
  (cond [(= n 0) '(())]
        [(= n (length l)) (list l)]
        [(null? l) '()]
        [(append (map (lambda (l1) (cons (car l) l1))
                      (choose (- n 1) (remove (car l) l)))
                 (choose n (cdr l)))]))

(define (takewhile p l)
  (cond [(or (null? l) (not (p (car l)))) '()]
        [else (cons (car l) (takewhile p (cdr l)))]))
```

```
(define (dropwhile p l)
  (cond [(or (null? l) (not (p (car l)))) l]
        [else (dropwhile p (cdr l))]))

; More functions on lists
(define (queens n)  solution of n X 8 queens problem
 (if (= n 0) `(())
     (append* (map extend-board (queens (- n 1))))))

(define (extend-board board)
 (define (extend-board-helper posns)
   (cond [(null? posns) `()]
         [(safe-board? (car posns) board)
              (cons (append board (list (cdar posns)))
                   (extend-board-helper (cdr posns)))]
         [else  (extend-board-helper (cdr posns))]))

 (extend-board-helper (generate-positions (+ 1 (length board)))))

(define (alltrue? p l)
 (foldr (lambda (x y) (and (p x) y)) #t l))

(define (safe-board? posn board)
 (let* ([full-board (expand board)])
   (andmap (lambda (posn1) (safe-pos? posn posn1))  full-board)))

(define (safe-pos? posn posn1)
 (let*([x (car posn)]
       [y (cdr posn)]
       [x1 (car posn1)]
       [y1 (cdr posn1)])
   (not (or (= y y1)
            (= (abs (- x x1)) (abs (- y y1)))))))

(define (zip l1 l2)
 (if (or (null? l1) (null? l2)) `()
     (cons (cons (car l1) (car l2)) (zip (cdr l1) (cdr l2)))))

(define (generate-positions n)
 (zip (make-list 8 n) (range 1 9)

(define (expand board) (zip (range 1 9) board))
```

Introduction to tree processing functions. Assume that the file trees-definition-and-example.rkt contains the following code

```
#lang racket

(provide t1 print-node (struct-out node) (struct-out nulltree)
)
(struct node (val ltree rtree) #:transparent)
(struct nulltree () #:transparent)

 Now the following functions get defined automatically
 The constructors node and nulltree
 The selectors node-val, node-ltree, node-rtree
 The predicate node? nulltree?

(define t4 (node 2
                     (node 1 (nulltree) (nulltree))
                          (node 3 (nulltree) (nulltree))))
(define t6 (node 6 (nulltree) (nulltree)))
(define t7 (node 2  (node 7 (nulltree) (nulltree))
                     (node 9 (nulltree) (nulltree))))
(define t8 (node 8 (node 7 (nulltree) (nulltree))
                     (node 9 (nulltree) (nulltree))))
(define t10 (node 12 (node 11 (nulltree) (nulltree))
                      (node 13 (nulltree) (nulltree))))
(define t11 (node 16 (node 15 (nulltree) (nulltree))
                      (node 17 (nulltree) (nulltree))))
(define t9 (node 14 t10 t11))
(define t3 (node 10  t8 t9))
(define t5 (node 5 (nulltree) (nulltree)))
(define t2 (node 4  t4 t5))
(define t1 (node 6 t2 t3))
```

```
my own printing routine
(define (print-nulltree)
  (begin (newline)
     (print-nulltree-helper 0)
     (newline)))
```

```
(define (print-node t)
  (print-node-helper t 0))


(define (print-node-helper t indent)
  (begin (newline)
     (printblanks indent)
     (write-string "(node ")
            (printblanks 1 op)
            (display (node-val t))
            (newline op)
            (let ((t1 (node-ltree t)))
         (cond ((node? t1) (print-node-helper t1 (+ indent 2)))
            ((nulltree? t1)  (print-nulltree-helper (+ indent 2)))))
            (let ((t1 (node-rtree t)))
         (cond ((node? t1) (print-node-helper t1 (+ indent 2)))
            ((nulltree? t1)  (print-nulltree-helper (+ indent 2)))))
     (write-char #\))
     (newline op)
     ))

(define (print-nulltree-helper indent)
  (begin (printblanks 1 )
    (write-string "N")))

(define (printblanks indent)
  (if (= indent 0) `()
          (begin
    (write-char #\space )
           (printblanks (- indent 1)))))
```

And assume that the file BinarySearchTrees.rkt has the following definitions. This is an example of modular programming.

```
#lang racket

(require "trees-definition-and-example.rkt")



(define (flatten t)
 (if (node? t) (append (flatten (node-ltree t))
                       (list (node-val t))
                       (flatten (node-rtree t)))
     `()))

(define (search x t)
 (cond [(nulltree? t) #f]
       [(= x (node-val t)) #t]
       [(< x (node-val t)) (search x (node-ltree t))]
       [else (search x (node-rtree t))]))

(define (insert x t)
 (cond [(nulltree? t) (node x (nulltree) (nulltree))]
       [(= x (node-val t)) t]
       [(< x (node-val t)) (node (node-val t) (insert x (node-ltree
t)) (node-rtree t))]
       [else (node (node-val t) (node-ltree t) (insert x (node-rtree
t)))]))

(define (delete x t)
 (cond [(nulltree? t) t]
       [(= x (node-val t)) (join (node-ltree t) (node-rtree t))]
       [(< x (node-val t)) (node (node-val t) (delete x (node-ltree
t))
                                 (node-rtree t))]
       [else (node (node-val t) (node-ltree t) (delete x (node-rtree
t)))]))
```

```
(define (join t1 t2)

 (define (rightmost t)
   (cond [(nulltree? (node-rtree t)) (node-val t)]
         [else (rightmost (node-rtree t))]))

 (define (readjust t)
   (cond [(nulltree? (node-rtree t)) (node-ltree t)]

 (cond [(null? t1) t2]
       [(null? t2) t1)]
       [else (node (rightmost t1) (readjust t1) t2)]))

(print-node (delete 14 t))  will print t with 14 deleted
```

Huffman code

The running example that we shall take has messages having the following characters with the corresponding frequencies: A-9, B-6, C-5, D-4, E-3, F-3, G-2, H-1.

A possible coding scheme for the symbols is

```
A - 10
B - 00
C - 110
D - 011
E - 010
F - 1111
G - 11101
H - 11100
```

To represent this coding scheme as a tree, we define the following structure:

```
(struct bnode (ltree rtree) #:transparent)
(struct leaf (val) #:transparent)
```

In terms of this structure let us encode the huffman tree for the code

```
(define example-huffman
  (bnode
   (bnode (leaf 'B) (bnode (leaf 'E) (leaf 'D)))
   (bnode (leaf 'A) (bnode (leaf 'C) (bnode
                                    (bnode (leaf 'H) (leaf 'G))
                                    (leaf 'F))))))
```

```
(define (decode t l)
  (define (decode-helper t1 l)
    (cond [(leaf? t1) (cons (leaf-val t1)
```

```
                            (decode t l))]
          [(null? l) (error "Ill formed input string")]
          [(= (car l) 0) (decode-helper (bnode-ltree t1) (cdr l))]
          [(= (car l) 1) (decode-helper (bnode-rtree t1) (cdr l))]))
  (if (null? l) `()
      (decode-helper t l)))
```

```
(decode example-huffman '(0 1 0 0 0 0 0 0 0 0 0 0 ))
```

The encode function does just the opposite. It takes a tree and a list of characters and converts into a list of bits. To do this it first converts the tree into a lookup table and searches in the table. For the tree above it generates the following table:

```
  ((A 1 0) (B 0 0) (C 1 1 0) ...  (H 1 1 1 0 0))
```

transform converts the tree into a lookup table

```
(define (transform t)
  (define (transform-helper l t)
    (cond ((leaf? t) (list (cons (leaf-val t) (reverse l))))
          (else (huffmerge (transform-helper (cons 0 l)  (bnode-ltree
t))
                           (transform-helper (cons 1 l)  (bnode-rtree
t))))))
  (transform-helper `() t))
```

```
(define (huffmerge l1 l2)
  (cond ((null? l1) l2)
          ((null? l2) l1)
          ((<= (length (car l1))
            (length (car l2)))
           (cons (car l1)
```

```
              (huffmerge (cdr l1) l2)))
        (else (cons (car l2)
                   (huffmerge l1 (cdr l2))))))
```

The function encode can now be written as:

```
(define tbl (transform example-huffman))
```

```
(define (encode l)
  (define (lookup sym tbl)
    (cond  [(null? tbl) (error "unknown symbol")]
           [(eq? sym (caar tbl)) (cdar tbl)]
           [else (lookup sym (cdr tbl))]))
  (append* (map (lambda (sym) (lookup sym tbl)) l)))
```

Now we show how the Huffman tree is obtained from a list giving the relative frequencies of occurrence of different characters. We also assume that the list is sorted in ascending order of the frequencies. This means that in our case the following list could be given: ((H.1) (G.2) (F.3) (E.3) (D.4) (C.5) (B.6) (A.9)).

first let us convert the initial list into a list of (binary-tree frequency) pairs.

```
(define (convert-initial-list l)
  (map (lambda (x) (cons (leaf (car x)) (cdr x))) l))
```

```
(define initial-list (list (cons 'H 1) (cons 'G  2)
                           (cons 'F  3) (cons 'E  3)
                           (cons 'D 4) (cons 'C  5)
                           (cons 'B  6) (cons 'A  9)))
```

Now given an initial list of (binary-tree weight) pairs, we should repeatedly do the following till we get a singleton list:

a. combine the first two trees in the list
b. insert the combination in the right place

The function combine combines two pairs (t1 w1) and (t2 w2) and makes the pair (t w1+w2), where t has t1 and t2 as its subtrees.

```
(define (combine p1 p2)
  (cons (bnode (car p1) (car p2))
        (+ (cdr p2) (cdr p1))))
```

a function insert which, given a (binary-tree weight) pair and a list of such pairs, insert the tree in the right place

```
(define (insert p l)
  (append (takewhile (lambda (x) (< (cdr x) (cdr p))) l)
          (list p)
          (dropwhile (lambda (x) (< (cdr x) (cdr p))) l)))


(define (dropwhile p l)
  (if (null? l) `()
      (if (p (car l)) (dropwhile p (cdr l)) l)))


(define (takewhile p l)
  (if (null? l) `()
      (if (p (car l)) (cons (car l) (takewhile p (cdr l)))
          `())))
```

combine the functions above into one called combine-and-insert

```
(define (combine-and-insert l)
  (insert (combine (car l) (cadr l)) (cddr l)))
```

we have to do combine-and-insert till the list becomes a singleton

```
(define (singleton? l) (null? (cdr l)))
```

Now let us write a function which applies a function f repeatedly till a condition is met

```
(define (until f p x)
  (if (p x) x (until f p (f x))))
```

Now we can write the function maketree which converts an initial list of (char weight) pairs to a huffman tree

```
(define (huffman l)
  (caar (until  combine-and-insert
                singleton?
                (convert-initial-list l))))
```