2.1

```
#lang racket

(define (make-rat n d)
  (let ((g (gcd n d))
        )
    (if  (< d 0) (make-rat (* -1 n) (* -1 d))

(cons (/ n g) (/ d g
    )))))


(define numer car)
(define denom cdr)

(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

2.2 and 2.3

```
#lang racket
(define (make-segment first-point second-point)
  (cons first-point second-point))

(define (make-point x y)
  (cons x y))

(define (x-point point)
  (car point))

(define (y-point point)
  (cdr point))

(define (start-segment segment)
  (car segment))
(define (end-segment segment)
  (cdr segment))

(define (print-point p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))

(define (square x) (* x x))

(define (length segment)
  (sqrt (+ (square (- (x-point (start-segment segment)) (x-point
(end-segment segment))))  (square (- (y-point (start-segment
segment)) (y-point (end-segment segment)))))))

(define (midpoint-segment segment)
  (let ((s1 (start-segment segment))
        (s2 (end-segment segment)))
       (let ((x1 (x-point s1))
       (y1 (y-point s1))
        (x2 (x-point s2))
       (y2 (y-point s2)))

  (make-point (/ (+ x1 x2) 2) (/ (+ y1 y2) 2)))))

(define (area rect)
  (* (width-size rect) (long-size rect)))

(define (perimeter rect)
  (* (+ (width-size rect) (long-size rect)) 2))

(define (make-rect long-segment width-segment)
  (cons long-segment width-segment))

(define (long rect)
  (car rect))

(define (width rect)
  (cdr rect))

(define (width-size rect)
  (length (width rect)))

(define (long-size rect)
  (length (long rect)))
```

2.4

```
#lang racket
(define (conss x y)
  (lambda (m) (m x y))

  )

(define (carr z)
  (z (lambda (p q) p)))

(define (cdrr z)
  (z (lambda (p q) q)))
```

2.5

```
#lang racket
(define (cons a b)
  (* (expt 2 a) (expt 3 b)))

(define (devide-count total num)
  (if (not (= (remainder total num) 0)) 0 (+ 1 (devide-count (/ total num) num))))

(define (car x)
  (devide-count x 2))

(define (cdr x)
   (devide-count  x 3))
```

2.6

```
(define one (lambda (f)
(lambda (x) (f x))))
(define two (lambda (f)
(lambda (x) (f (f x)))))
```

2.7 2.8 2.9

```
#lang racket

(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (upper-bound y))
                 ))
(define (sub-interval x y)
  (add-interval x (make-interval (* -1 (lower-bound y)) (* -1 (upper-
bound y)))))

(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
    (p2 (* (lower-bound x) (upper-bound y)))
    (p3 (* (upper-bound x) (lower-bound y)))
    (p4 (* (upper-bound x) (upper-bound y))))
  (make-interval (min p1 p2 p3 p4) (max p1 p2 p3 p4))))

(define (div-interval x y)
  (div-interval x
                (make-interval (/ 1.0 (upper-bound y))
                               (/ 1.0 (lower-bound y)))))

(define (make-interval a b) (cons a b))

(define (upper-bound interval) (cdr interval))
(define (lower-bound interval) (car interval))

(define (width interval)
  (/ (+ (upper-bound interval) (lower-bound interval)) 2))
```

(width (add-interval x))

(with (make-interval (+ (lower-bound x) (lower-bound y)) (+ (upper-bound x) (upper-bound y))

((+ (lower-bound x) (lower-bound y)) (+ (upper-bound x) (upper-bound y)))/2)

(+ (width x) (width y))

$(t_1+k_1)(t_1-k_1) \Rightarrow$

$(t_2+k_2)(t_2-k_2)$

$(t_1+k_1)(t_2+k_2) = t_1t_2+k_1k_2+k_1t_2+k_2t_1$

$(t_1-k_1)(t_2-k_2) = t_1t_2+k_1k_2-k_1t_2-k_2t_1$

width $= \frac{t_1t_2+k_1k_2-k_1t_2-k_2t_1}{t_1t_2+k_1k_2} = 1-\frac{k_1t_2+k_2t_1}{t_1t_2+k_1k_2}$

$= \frac{\alpha_1 t_1 t_2 + \alpha_2 t_1 t_2}{t_1t_2+\alpha_1\alpha_2 t_1t_2} \simeq \alpha_1+\alpha_2$