Exercise 2.1

make-rat that handles both positive and negative arguments

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cond ((> (* n d) 0) (cons (/ (abs n) g) (/ (abs d) g)))
      (else (cons (/ (- 0 (abs n)) g) (/ (abs d) g))))))

(define x (make-rat 1 -2))
(car x)
;; -1
(cdr x)
;; 2
```

Exercise 2.2

segment and print midpoint

```
(define (make-point x y)
  (cons x y))
(define (x-point x)
  (car x))
(define (y-point x)
  (cdr x))
(define (make-segment start end)
  (cons start end))
(define (start-segment seg)
  (car seg))
(define (end-segment seg)
  (cdr seg))
(define (midpoint-segment seg)
  (make-point (/ (+ (x-point (start-segment seg)) (x-point (end-segment seg))) 2)
          (/ (+ (y-point (start-segment seg)) (y-point (end-segment seg))) 2)))
(define (print-point p) (newline) (display "(") (display (x-point p)) (display ",") (display (y-point p)) (display ")"))
 ;; Testing
(define seg (make-segment (make-point 2 3)
                          (make-point 10 15)))

(print-point (midpoint-segment seg))
;; result: (6,9)
```

Exercise 2.3

rectangle class with area and perimeter implement

```
(define (make-point x y)
  (cons x y))
(define (x-point point)
  (car point))
(define (y-point point)
  (cdr point))
(define (make-segment pointx pointy)
  (cons pointx pointy))
(define (make-rectangle pointx pointy)
  (cons pointx pointy))
(define (make-rectangle pointx pointy)
  (make-segment pointx pointy))
(define (start-rect rect)
  (car rect))
(define (end-rect rect)
  (cdr rect))

;; barrier start end
(define (rect-width rect)
  (- (x-point (end-rect rect)) (x-point (start-rect rect))))
(define (rect-height rect)
  (- (y-point (end-rect rect)) (y-point (start-rect rect))))

;; barrier width height
(define (perimeter rect)
  (* 2 (+ (rect-width rect)
      (rect-height rect))))
(define (area rect)
  (* (rect-width rect)
      (rect-height rect)))
(define rect (make-rectangle (make-point 2 3)
                 (make-point 10 15)))
(perimeter rect)
;; result: 40
(area rect)
;; result: 96
```

Exercise 2.4

pair representation

```
(define (cons x y)
  ;; a procedure (m x y)
  (lambda (m) (m x y)))
```

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

(car (cons 1 2))
;; result: 1
;; (car (cons x y))
;; ((car m) (m x y))
;; ((car ((lambda (p q) p) x y)))
;; =>x

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

Exercise 2.5

```
(define (cons a b)
  (* (expt 2 a) (expt 3 b)))
(define (car c)
  (cond ((= (remainder c 3) 0) (car (/ c 3)))
    (else (/ (log c) (log 2)))))
(define (cdr c)
  (cond ((= (remainder c 2) 0) (cdr (/ c 2)))
    (else (/ (log c) (log 3)))))

(car (cons 2 3))
;; result: 2
(cdr (cons 2 3))
;; result: 3
```

Exercise 2.6

Church numerals

```
(define
  zero
  (lambda (f) (lambda (x) x)))
(define
  (add-1 n)
  (lambda (f)
    (lambda (x) (f ((n f) x)))))
;; (add-1 zero)
;; (lambda (f) (lambda (x) (f ((zero f) x))))
;; (lambda (f) (lambda (x) (f x)))
(define
  one
```

```
  (lambda (f) (lambda (x) (f x))))
;; (add-1 one)
;; (lambda (f) (lambda (x) (f ((one f) x))))
;; (lambda (f) (lambda (x) (f ((lambda (x) (f x)) x))))
;; (lambda (f) (lambda (x) (f (f x))))
(define
  two
  (lambda (f) (lambda (x) (f (f x)))))
;; (+ one two)
;; definition of the addition: use n2 as n1's parameter
(define (+ n1 n2)
  (n1 n2))
```

Exercise 2.7, 2.8, 2.10, 2.11, 2.12, 2.13, 2.14

```
(define (make-interval a b) (cons a b))
(define (upper-bound c)
  (max (car c) (cdr c)))
(define (lower-bound c)
  (min (car c) (cdr c)))
(define (add-interval x y) (make-interval (+ (lower-bound x) (lower-
    bound y)) (+ (upper-bound x) (upper-bound y))))
(define (sub-interval x y)
  (make-interval (- (lower-bound x) (upper-bound y)) (- (upper-bound x)
       (lower-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))))
;; Exercise 2.11
(define (mul-interval x y)
  (let ((xl (lower-bound x))
    (xh (upper-bound x))
    (yl (lower-bound y))
    (yh (upper-bound y)))
```

```
    (cond ((and (< xh 0) (< yh 0))
       (make-interval (* xh yh) (* xl yl)))
      ((and (> xl 0) (> yl 0))
       (make-interval (* xl yl) (* xh yh)))
      ((and (< xh 0) (> yl 0))
       (make-interval (* xl yh) (* xh yl)))
      ((and (> xl 0) (< yh 0))
       (make-interval (* xh yl) (* xl yh)))
      ((and (< xh 0) (< yl 0) (> yh 0))
       (make-interval (* xl yh) (* xh yl)))
      ((and (< xl 0) (> xh 0) (< yl 0) (> yh 0))
       (make-interval (min (* xl yh) (* xh yl)) (max (* xl yl) (* xh yh
          ))))
      ((and (< xl 0) (> xh 0) (< yh 0))
       (make-interval (* xh yl) (* xl yl)))
      ((and (< xl 0) (> xh 0) (> yl 0))
       (make-interval (* xl yh) (* xh yh)))
      ((and (> xl 0) (< yl 0) (> yh 0))
       (make-interval (* xh yl) (* xh yh))))))
(define (print-interval x)
  (newline)
  (display "[")
  (display (lower-bound x))
  (display ",")
  (display (upper-bound x))
  (display "]"))
(define
  (div-interval x y)
  (if (< (* (lower-bound y) (upper-bound y)) 0)
      (error "the second argument span 0" y)
      (mul-interval
       x
       (make-interval
    (/ 1.0 (upper-bound y))
    (/ 1.0 (lower-bound y))))))
;; Exercise 2.12
(define (make-center-width c w) (make-interval (- c w) (+ c w)))
(define (center i) (/ (+ (lower-bound i) (upper-bound i)) 2))
(define (width i) (/ (- (upper-bound i) (lower-bound i)) 2))
(define (make-center-percent c p) (let ((w (* c p))) (make-center-width
    c w)))
(define (percent interval)
  (let ((w (width interval))
    (c (center interval)))
    (/ w c)))
;; (div-interval (make-interval 2 3) (make-interval -1 1))
;; (make-interval -1 1)
;; (mul-interval (make-interval 2 3) (make-interval -1 1))
;; result: (-3, 3)
;; (percent (make-center-percent 100 0.01))
;; result: .01
```

```
;; Exercise 2.14
(define A (make-center-percent 100 0.00001))
(define B (make-center-percent 0.000000000000000000000001 0.00001))
(define A/A (div-interval A A))
(percent A/A)
(center A/A)
;; 0.00002, 1.0000000002
(define A/B (div-interval A B))
(percent A/B)
(center A/B)
;; 0.00002,2.0000000004
(define
  (par1 r1 r2)
  (div-interval
   (mul-interval r1 r2)
   (add-interval r1 r2)))
(define
  (par2 r1 r2)
  (let ((one (make-interval 1 1)))
    (div-interval
     one
     (add-interval
      (div-interval one r1)
      (div-interval one r2)))))

(define par1AB (par1 A B))
(define par2AB (par2 A B))
(percent par1AB)
;; 0.00002999
(percent par2AB)
;; 0.00001
```

Exercise 2.9

$$\begin{aligned} x+y &= [lower(x)+lower(y), upper(x)+upper(y)] \\ width(x+y) &= \frac{upper(x)+upper(y)-lower(x)-lower(y)}{2} \\ &= \frac{2*width(x)+2*width(y)}{2} \\ &= width(x)+width(y) \end{aligned}$$

$$\begin{aligned}
x - y &= [lower(x) - upper(y), upper(x) - lower(y)] \\
width(x - y) &= \frac{upper(x) + upper(y) - lower(x) - lower(y)}{2} \\
&= \frac{2 * width(x) + 2 * width(y)}{2} \\
&= width(x) + width(y)
\end{aligned}$$

$$[1, 2]/[3, 4] = [1/4, 4/1] = [0.25, 4] \neq width(x) + width(y) or width(x) - width(y)$$

Exercise 2.15

Yes, because par1 use R_1 and R_2 twice

Exercise 2.16

Because everytime use an interval can lead to some difference