```scheme
(load "chap1.scm")

;;Exercise 2.1
(define better-make-rat
  (lambda (x y)
    (let ((g (gcd x y)))
      (if(or (and (> x 0) (> y 0)) (and (< x 0) (< y 0)))
         (cons (abs (/ x g)) (abs (/ y g)))
         (cons (- (abs (/ x g))) (abs (/ y g)))))))
(define (numer x) (car x))
(define (denom x) (cdr x))

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

;;Exercise 2.2
(define (print-p p)
  (newline)
  (display "(")
  (display (x-point p))
  (display ",")
  (display (y-point p))
  (display ")"))

(define make-point
  (lambda (x y)
    (cons x y)))
(define x-point
  (lambda (point)
    (car point)))
(define y-point
  (lambda (point)
    (cdr point)))

(define make-segment
  (lambda (start-point end-point)
    (cons start-point end-point)))
(define start-segment
  (lambda (segment)
    (car segment)))
(define end-segment
```

```
  (lambda (segment)
    (cdr segment)))

(define mid-point
  (lambda (segment)
    (make-segment (/ (+  (x-point (start-segment segment))
                         (x-point (end-segment segment)))
                     2)
                  (/ (+ (y-point (start-segment segment))
                        (y-point (end-segment segment)))
                     2))))

;;Exercise 2.3
(define make-rect
  (lambda (width height)
    (cons width height)))
(define (width a-rect)
  (car a-rect))
(define (height a-rect)
  (cdr a-rect))
(define perimeter
  (lambda (a-rect)
    (* 2 (+ (width a-rect)
            (height a-rect)))))
(define area
  (lambda (a-rect)
    (* (width a-rect)
       (height a-rect))))
;;Version2 : using points representation

;;Exercise 2.4
(define (my-cons x y)
  (lambda (m) (m x y)))

(define (my-car z)
  (z (lambda (p q) p)))

;(my-car (my-cons 1 2))
(define (my-cdr z)
  (z (lambda (p q) q)))
;(my-cdr (my-cons 1 2))

;;Exercise 2.5
;;Because an Integer can and represent by (2^a)*(3^b)
```

```
(define my-cons
  (lambda (a b)
    (* (expt 2 a) (expt 3 b))))
(define my-car
  (lambda (a-pair)
    (define (loop input value)
      (if (= (remainder input 2) 0)
          (loop (/ input 2) (+ value 1))
          value))
    (loop a-pair 0)))
(define my-cdr
  (lambda (a-pair)
    (define (loop input value)
      (if (= (remainder input 3) 0)
          (loop (/ input 3) (+ value 1))
          value))
    (loop a-pair 0)))

;;Exercise 2.6
;丘齐数里，所有的数字都是两个参数的函数:
;零是 lambda s z . z
;一是 lambda s z . s z
;二是 lambda s z . s (s z)
;对任意一个数"n"，它的丘齐数都是一个函数。这个函数把它的第一个参数应用到第二个参数上n次。

;加法函数 plus(m,n) = m + n 利用了恒等式 f(m + n)(x) = fm(fn(x))。
;
;plus ≡ λm.λn.λf.λx. m f (n f x)
;后继函数 succ(n) = n + 1 β-等价于 (plus 1)。
;
;succ ≡ λn.λf.λx. f (n f x)
;乘法函数 times(m,n) = m * n 利用了恒等式 f(m * n) = (fm)n。
;
;mult ≡ λm.λn.λf. n (m f)
;指数函数 exp(m,n) = mn 由 Church 数定义直接给出。
;
;exp ≡ λm.λn. n m

(define zero
  (lambda (f)
    (lambda (x)
      x)))
;;zero是一个函数，这个函数接受参数，返回参数，不做任何处理
```

```
;;use substitution model to evaluate (add-1 zero)
;;(add-1 one) to get two
;;(add-1 two) to get three

(define one
  (lambda (f)
    (lambda (x)
      (f x))))

(define two
  (lambda (f)
    (lambda (x)
      (f (f x)))))

(define (add-1 n)
  (lambda (f)
    (lambda (x)
      (f ((n f) x)))))
;;operate one more time than before

(define (church-add n m)
  (lambda (f)
    (lambda (x)
      ((((n add-1) m) f) x))))
;;using repeated add-1
(define (my-church-add n m)
  (lambda (f)
    (lambda (x)
      ((m f) ((n f) x)))))

(define (church-mul n m)
  (lambda (f)
    (n (m f))))

(define (church-expt m n)
  (n m))

(define f
  (lambda (x)
    (cons "a" x)))

;(((add-1 two) f) '())
;(((my-church-add one two) f) '())
;A better test version: (define (inc x) (+ 1 x))
```

```
(define (inc x) (+ x 1))
(define (check n)
  ((n inc) 0))

;;Extended Exercise: Interval Arithmetic
(define (add-interval x y)
  (make-interval (+ (lower-bound x) (lower-bound y))
                 (+ (upper-bound x) (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)
  (mul-interval x
                (make-interval (/ 1.0 (upper-bound y))
                               (/ 1.0 (lower-bound y)))))
;;Exercise 2.7
(define make-interval
  (lambda (low up)
    (cons low up)))
(define (upper-bound a)
  (cdr a))
(define (lower-bound a)
  (car a))
;;Exercise 2.8
(define (sub-interval x y)
  (make-interval (- (upper-bound x) (lower-bound y))
                 (- (lower-bound x) (upper-bound y))))

;;Exercise 2.9
;For interval (4 6) and (7 11)   width-add= (6-4)/2 + (11-7)/2 width-sub=width-
add
;For mul and div doesn't stand
;概率与统计: E(X+Y)=E(x)+E(Y)
;X与Y相互独立时: E(XY)=E(X)E(Y)

;;Exercise 2.10
(define check-div-interval
  (lambda (a-interval b-interval)
    (if (and(> (upper-bound b-interval) 0)
```

```
          (< (lower-bound b-interval) 0))
        (display "Error occured!")
        (div-interval a-interval b-interval))))

;;Exercise 2.11
(define mul-interval
  (lambda (x y)
    (cond ((>= (lower-bound x) 0)
           (cond ((>= (lower-bound y) 0)
                  (make-interval (* (lower-bound x) (lower-bound y)) (* (upper-
bound x) (upper-bound y))))
                 ((<= (upper-bound y) 0)
                  (make-interval (* (upper-bound x) (lower-bound y)) (* (lower-
bound x) (upper-bound y))))
                 (else
                  (make-interval (* (upper-bound x) (lower-bound y)) (* (lower-
bound x) (upper-bound y))))))
          ((<= (upper-bound x) 0)
           (cond ((>= (lower-bound y) 0)
                  (make-interval (* (lower-bound x) (upper-bound y)) (* (upper-
bound x) (lower-bound y))))
                 ((<= (upper-bound y) 0)
                  (make-interval (* (lower-bound x) (lower-bound y)) (* (upper-
bound x) (upper-bound y))))
                 (else
                  (make-interval (* (lower-bound x) (upper-bound y)) (* (lower-
bound x) (lower-bound y))))))
          (else
           (cond ((>= (lower-bound y) 0)
                  (make-interval (* (lower-bound x) (upper-bound y)) (* (upper-
bound x) (upper-bound y))))
                 ((<= (upper-bound y) 0)
                  (make-interval (* (upper-bound x) (lower-bound y)) (* (lower-
bound x) (lower-bound y))))
                 (else
                  (make-interval (min (* (lower-bound x) (upper-bound y))
                                      (* (upper-bound x) (lower-bound y)))
                                 (max (* (lower-bound x) (lower-bound y))
                                      (* (upper-bound x) (upper-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
  (lambda (center percent)
    (let ((balance (/ (* center percent) 100)))
      (make-interval (- center balance) (+ center balance)))))

(define center
  (lambda (a-interval)
    (/ (+ (lower-bound a-interval) (upper-bound a-interval))
       2)))
(define percent
  (lambda (a-interval)
    (let ((value (- (upper-bound a-interval) (center a-interval))))
      (* (/ value (center a-interval)) 100))))

;;Exercise 2.13
;interval A: (a-a*p/100, a+a*p/100)
;interval B: (b-b*p/100, b+b*p/100)
;A*B = ab * (1 + pq/10000) ± (p+q)/100
;for small p and q, pq/1000=0
;A*B= ab±(p+q)/100

;;Exericse 2.14
(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 x (make-interval 9.5 10.5)) ;; 10.0 ±0.5  = 10.0 ±5%
(define y (make-interval 5 6))      ;;  6.5 ±0.5  =  6.5 ±7.69%
(define z (make-interval 2 2.5))    ;;  2.25±0.25 =  2.25±11.1%
(define A (make-interval 8.999 9.001))
(define B (make-interval 7.999 8.000))
; Lem is right
;(par1 x y)
;(par2 x y)   the width of par2 is smaller
```

```
;(div-interval A A)
;(0.9997778024663928 . 1.0002222469163238)

;;Exercise 2.15 and 2.16...

;(par1 x y)
;(2.878787878787879 . 4.344827586206897)
;(par2 x y)
;(3.2758620689655173 . 3.8181818181818181819)

;par2 produce a smaller width...
;there is some difference between interval-arithmetic  and it's equivalent
algebraic expressions

;;Exercise 2.17
(define my-last-pair
  (lambda (a-pair)
    (define (loop list)
      (if (empty? (cdr list))
          (cons (car list) '())
          (loop (cdr list))))
    (loop a-pair)))
;;Exercise 2.18
(define my-reverse
  (lambda (a-list)
    (define (loop list list-now)
      (if (empty? list)
          list-now
          (loop (cdr list) (cons (car list) list-now))))
    (loop a-list '())))

;;Exercise 2.19
(define first-denomination
  (lambda (value-list)
    (car value-list)))
(define except-first-denomination
  (lambda (value-list)
    (cdr value-list)))

(define no-more?
  (lambda (value-list)
    (empty? value-list)))

(define (cc amount coin-values)
```

```
  (cond ((= amount 0) 1)
        ((or (< amount 0) (no-more? coin-values)) 0)
        (else
         (+ (cc amount
                (except-first-denomination coin-values))
            (cc (- amount
                   (first-denomination coin-values))
                coin-values)))))
;;Exercise 2.20
(define (same-parity . w)
  (let ((first-even? (even? (car w))))
    (define (same-parity? item)
      (let ((this-even? (even? item)))
        (or (and this-even? first-even?)
            (and (not this-even?) (not first-even?)))))
    (define (loop items result)
      (cond ((empty? items) result)
            ((same-parity? (car items))
             (cons (car items) (loop (cdr items) result)))
            (else
             (loop (cdr items) result))))
    (loop w '())))

;(define nil '())
;;Exercise 2.21
;Version 1
(define (square-list items)
  (if (null? items)
      '()
      (cons (square (car items)) (square-list (cdr items)))))

;Version 2
(define (my-map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))

(define (square-list items)
  (my-map square items))

;;Exercise 2.22
;This is the correct version
(define (square-list items)
```

```
  (define (iter things answer)
    (if (null? things)
        answer
        (cons (square (car things))
              (square-list (cdr things)))))
  (iter items '()))

;;Exercise 2.23
(define my-for-each
  (lambda (proc items)
    (define (loop things)
      (if (null? things)
          (display "\nOK")
          (begin
            (proc (car things))
            (loop (cdr things)))))
    (loop items)))

;;Exercise 2.24
;(1 (2 (3 4)))

;;Exercise 2.25
;(cadaddr '(1 3 '(5 7) 9))
;(car (cdaddr '( 1 3 (5 7) 9)))
;(caar '((7)))
;(car (cdr (car (cdr (car (cdr (car (cdr (cadadr '(1 (2 (3 (4 (5 (6
7))))))))))))))

;;Exercise 2.26
(define x (list 1 2 3))
(define y (list 4 5 6))
;(append x y)
;(cons x y)
;(list x y)
;(1 2 3 4 5 6)
;((1 2 3) 4 5 6)
;((1 2 3) (4 5 6))

;;Exercise 2.27
(define x (list (list 1 2) (list 3 4)))
(define deep-reverse
  (lambda (items)
    (define (loop things now)
      (cond ((null? things) now)
```

```scheme
          ((pair? (car things))
           (let ((temp (reverse (car things))))
             (loop (cdr things) (cons temp now))))
          (else
           (loop (cdr things) (cons (car things) now)))))
    (loop items '())))

;;Exercise 2.28
(define fringe
  (lambda (tree)
    (cond ((null? tree) '())
          ((pair? (car tree))
           (append (fringe (car tree)) (fringe (cdr tree))))
          (else
           (cons (car tree) (fringe (cdr tree)))))))

;;Exercise 2.29
(define (make-mobile left right)
  (cons left right))

(define (make-branch length structure)
  (cons length structure))
;a
(define (left-branch mobile)
  (car mobile))
(define (right-branch mobile)
  (cadr mobile))

(define (branch-length branch)
  (car branch))
(define (branch-structure branch)
  (cadr branch))
;b
(define (total-weight mobile)
  (define (branch-weight branch)
    (if(not (pair? (branch-structure branch)))
       (branch-structure branch)
       (total-weight (branch-structure branch))))
  (+ (branch-weight (left-branch mobile))
     (branch-weight (right-branch mobile))))
;c   Design a predicate that tests whether a binary mobile is balanced.
(define (branch-weight branch)
  (if(not (pair? (branch-structure branch)))
     (branch-structure branch)
```

```scheme
    (total-weight (branch-structure branch))))

(define (structure-is-mobile? branch)
  (pair? (branch-structure branch)))

(define (troque branch)
  (* (branch-length branch)
     (branch-weight branch)))

(define (balanced-branch? branch)
  (if(structure-is-mobile? branch)
     (balanced-mobile? (branch-structure branch))
     #t))

(define (balanced-mobile? mobile)
  (let ((left (left-branch mobile))
        (right (right-branch mobile)))
    (if (= (troque left) (troque right))
        (and (balanced-branch? left)
             (balanced-branch? right))
        #f)))
;d
;change
(define (make-mobile left right)
  (cons left right))
(define (make-branch length structure)
  (cons length structure))

(define (right-branch mobile)
  (cdr mobile))
(define (branch-structure branch)
  (cdr branch))

;;Exercise 2.30
(define square-tree
  (lambda (tree)
    (cond ((null? tree) '())
          ((not (pair? tree)) (* tree tree))
          (else (cons (square-tree (car tree))
                      (square-tree (cdr tree)))))))

(define (square-tree tree)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
```

```scheme
            (square-tree sub-tree)
            (* sub-tree sub-tree)))
      tree))

;;Exercise 2.31
(define (tree-map proc tree)
  (map (lambda (sub-tree)
         (if(pair? sub-tree)
            (tree-map proc sub-tree)
            (proc sub-tree)))
      tree))
;;Exercise 2.32
(define (subsets s)
  (if (null? s)
      (list '())
      (let ((rest (subsets (cdr s))))
        (append rest (map (lambda(a-list)
                            (cons (car s) a-list))
                          rest)))))
;;For example: (subsets (list 1 2 3))
;(())
;((cons 3 ()) ()) ==> ((3) ())
;((cons 2 (3)) (cons 2 ()) (3) ()) ==> ((2 3) (2) (3) ())
;((cons 1 (2 3)) (cons 1 (2)) (cons 1 (3)) (cons 1 ()) (2 3) (2) (3) ()) ==>((1
2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())

;;Exercise 2.33
(define (accumulate op initial sequences)
  (if (null? sequences)
      initial
      (op (car sequences)
          (accumulate op initial (cdr sequences)))))

(define (my-map p sequences)
  (accumulate
   (lambda (x y)
     (cons (p x) y))
   '()
   sequences))

(define (my-append seq-a seq-b)
  (accumulate
   cons
   seq-b
```

```
  seq-a))

(define (my-length seq)
  (accumulate
   (lambda (x y)
     (+ y 1))
   0
   seq))

;;Exercise 2.34
(define (horner-eval x coefficient-sequence)
  (accumulate
   (lambda (this-coeff higher-terms)
     (+ (* higher-terms x)
        this-coeff))
   0
   coefficient-sequence))

;;Exercise 2.35
(define (count-leaves tree)
  (accumulate
   (lambda (x y)
     (+ x y))
   0
   (map (lambda (node)
          (cond ((null? node) 0)
                ((pair? node) (count-leaves node))
                (else
                 1)))
        tree)))

;;Exercise 2.36
(define (accumulate-n op init seqs)
  (if (null? (car seqs))
      '()
      (cons (accumulate op init (map car seqs))
            (accumulate-n op init (map cdr seqs)))))

;;Exercise 2.37
(define (dot-product v w)
  (accumulate + 0 (map * v w)))

(define (matrix-*-vector m v)
  (map
```

```
   (lambda (a-vector)
     (dot-product a-vector v))
   m))

(define (transpose mat)
  (accumulate-n
   cons
   '()
   mat))

(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map
     (lambda (a-vector)
       (map (lambda (cols-vector)
              (dot-product cols-vector a-vector))
            cols))
     m)))
;;Exercise 2.38
(define fold-right accumulate)

(define (fold-right op initial sequences)
  (if (null? sequences)
      initial
      (op (car sequences)
          (accumulate op initial (cdr sequences)))))

(define (fold-left op initial sequences)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequences))
;;(fold-right / 1 (list 1 2 3)) ==>   3/2
;;(fold-left / 1 (list 1 2 3))    ==> 1/6
;;(fold-right list '() (list 1 2 3)) ==>   (1 (2 (3 ())))
;;(fold-left list '() (list 1 2 3))  ==>   (((() 1) 2) 3)
;;Associative

;;Exercise 2.39
(define nil '())

(define (my-reverse sequences)
```

```
  (fold-right
   (lambda (x y)
     (append y (list x)))
   nil sequences))

(define (my-reverse sequences)
  (fold-left
   (lambda (x y)
     (cons y x))
   nil
   sequences))

;;Exercise 2.40
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))

(define (enumerate-interval low high)
  (define (loop index now)
    (if (> index high)
        now
        (loop (+ index 1)
              (append now (list index)))))
  (loop low nil))

(define (unique-pairs n)
  (flatmap
   (lambda (index)
     (map (lambda (x)
            (list index x))
          (enumerate-interval 1 index)))
   (enumerate-interval 1 n)))
;;Exercise 2.41
(define (ordered-one-triple n)
  (map
   (lambda (a-list)
     (cons n a-list))
   (unique-pairs (- n 1))))

(define (ordered-triples n)
  (flatmap
   ordered-one-triple
   (enumerate-interval 1 n)))

(define (S-triple n s)
```

```
  (define (Ok? a-triple)
    (= (+ (car a-triple) (cadr a-triple) (caddr a-triple))
       s))
  (let ((sequences (ordered-triples n)))
    (filter
     Ok?
     sequences)))

;;Exercise 2.42
(define (queens board-size)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
         (lambda (positions) (safe? k positions))
         (flatmap
          (lambda (rest-of-queens)
            (map (lambda (new-row)
                   (adjoin-position new-row k rest-of-queens))
                 (enumerate-interval 1 board-size)))
          (queen-cols (- k 1))))))
  (queen-cols board-size))

(define empty-board nil)

(define (adjoin-position new-row k rest-of-queens)
  (cons (list new-row k) rest-of-queens))

(define (safe? k positions)
  (let ((k-th (car positions)))
    (define (conflicts? a b)
      (let ((dx (abs (- (car a) (car b))))
            (dy (abs (- (cadr a) (cadr b)))))
        (cond ((= dx 0) #t)
              ((= dy 0) #t)
              ((= dx dy) #t)
              (else #f))))
    (define (Test pos)
      (cond((null? pos) #t)
           ((conflicts? (car pos) k-th) #f)
           (else
            (Test (cdr pos)))))
    (Test (cdr positions))))
```

```
;;Exercise 2.43
;速度会慢得很多，因为随着k值增大，rest-of-queens增大很多，

;;==============================Example:  A Picture Languange========================
;;=============================2.3 Symbolic Data============================

;;Exercise 2.53
;(list 'a 'b 'c)
;(list (list 'george))   ;(a b c)
;(cdr '((x1 x2) (y1 y2)))  ;((george))
;(cadr '((x1 x2) (y1 y2)))  ;((y1 y2)) (y1 y2)
;(pair? (car '(a short list))) ;#f
;(memq 'red '((red shoes) (blue socks)))  ;#f
;(memq 'red '(red shoes blue socks)) ;(red shoes blue socks)

;;Exercise 2.54
(define (my-equal? a b)
  (cond ((and (symbol? a) (symbol? b))
        (if (eq? a b)
            #t
            #f))
       ((and (number? a) (number? b))
        (= a b))
       ((and (null? a) (null? b)) #t)
       ((and (list? a) (list? b))
        (if (my-equal? (car a) (car b))
            (my-equal? (cdr a) (cdr b))
            #f))
       (else
        #f)))

;;Exercise 2.55
;(car '(quote abracadabra))


;;Example: Symbolic Differentiation

(define (variable? x) (symbol? x))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))

(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
```

```
       ((=number? a2 0) a1)
       ((and (number? a1) (number? a2)) (+ a1 a2))
       (else (list '+ a1 a2))))
(define (=number? exp num)
  (and (number? exp) (= exp num)))


(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
       ((=number? m1 1) m2)
       ((=number? m2 1) m1)
       ((and (number? m1) (number? m2)) (* m1 m2))
       (else (list '* m1 m2))))

(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s) (caddr s))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))

(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))

;;Exercise 2.56
(define (exponentiation? x)
  (and (pair? x) (eq? (car x) '**)))
(define (base p)
  (cadr p))
(define (exponent p)
  (caddr p))
(define (make-exponentiation base expt)
  (cond ((= expt 0) 1)
       ((= expt 1) base)
       (else
        (list '** base expt))))


(define (deriv exp var)
  (cond ((number? exp) 0)
       ((variable? exp)
        (if (same-variable? exp var) 1 0))
       ((sum? exp)
```

```scheme
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))
        ((exponentiation? exp)
         (make-product (make-product (exponent exp)
                                     (make-exponentiation (base exp) (- (exponent
exp) 1)))
                       (deriv (base exp) var)))
        (else
         (error "unknown expression type -- DERIV" exp))))

;;Exercise 2.57
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (addend s) (cadr s))
(define (augend s)
  (if (null? (cdddr s))
      (caddr s)
      (cons '+ (cddr s))))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
(define (multiplier p) (cadr p))
(define (multiplicand p)
  (if (null? (cdddr p))
      (caddr p)
      (cons '* (cddr p))))

;;Exericse 2.58
;;a
(define (sum? x)
  (and (pair? x) (eq? (cadr x) '+)))
(define (addend s)
  (car s))
(define (augend s)
  (caddr s))

(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
```

```
       ((=number? a2 0) a1)
       ((and (number? a1) (number? a2)) (+ a1 a2))
       (else (list a1 '+ a2))))

(define (product? x)
  (and (pair? x) (eq? (cadr x) '*)))
(define (multiplier p) (car p))
(define (multiplicand p)
  (caddr p))

(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
       ((=number? m1 1) m2)
       ((=number? m2 1) m1)
       ((and (number? m1) (number? m2)) (* m1 m2))
       (else (list m1 '* m2))))

(define (exponentiation? x)
  (and (pair? x) (eq? (cadr x) '**)))
(define (base p)
  (car p))
(define (exponent p)
  (caddr p))
(define (make-exponentiation base expt)
  (cond ((= expt 0) 1)
       ((= expt 1) base)
       (else
        (list base '** expt))))

;b ;;This is two harder, i will solve this latter =_=!!

;;Exercise 2.59
(define (element-of-set? x set)
  (cond ((null? set) false)
       ((equal? x (car set)) true)
       (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (if (element-of-set? x set)
     set
     (cons x set)))

(define (union-set set1 set2)
  (define (loop a-set set-now)
```

```
    (if(null? a-set)
       set-now
       (loop (cdr a-set)
             (adjoin-set (car a-set) set-now))))
  (loop set1 set2))

;;Exercise 2.60
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else
         (element-of-set? x (cdr set)))))
(define (adjoin-set x set)
  (cons x set))
(define (union-set set1 set2)
  (append set1 set2))

(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))

;;Exericse 2.61
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (cond ((< x (car set)) (cons x set))
        ((= x (car set)) set)
        (else
         (cons (car set) (adjoin-set x (cdr set))))))

;;Exercise 2.62
(define (union-set set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let((x1 (car set1))
              (x2 (car set2)))
```

```
          (cond ((= x1 x2) (cons x1 (union-set (cdr set1) (cdr set2))))
                ((< x1 x2) (cons x1 (union-set (cdr set1) set2)))
                (else
                 (cons x2 (union-set set1 (cdr set2))))))))))
;;Exercise 2.63
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree) (caddr tree))
(define (make-tree entry left right)
  (list entry left right))

(define (tree->list-1 tree)
  (if (null? tree)
      '()
      (append (tree->list-1 (left-branch tree))
              (cons (entry tree)
                    (tree->list-1 (right-branch tree))))))
;;order:  left--entry--right
(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
                      (cons (entry tree)
                            (copy-to-list (right-branch tree)
                                          result-list)))))
  (copy-to-list tree '()))

;;test
(define tree1
  (make-tree 1
             (make-tree 2
                        (make-tree 4 nil nil)
                        (make-tree 5 nil nil))
             (make-tree 3 (make-tree 6 nil nil)
                        (make-tree 7 nil nil))))
(define tree2
  (make-tree 1
             nil (make-tree 2
                            nil (make-tree 3
                                           nil (make-tree 4
                                                          nil (make-tree 5
                                                                         nil (make-tree 6
                                                                                        nil
```

```scheme
(make-tree 7 nil nil))))))))
;;the two version tree-list visit tree in the same order
;b: Version1 grows slowly

;;Exercise 2.64
(define (list->tree elements)
  (car (partial-tree elements (length elements))))

(define (partial-tree elts n)
  (if (= n 0)
      (cons '() elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
          (let ((left-tree (car left-result))
                (non-left-elts (cdr left-result))
                (right-size (- n (+ left-size 1))))
            (let ((this-entry (car non-left-elts))
                  (right-result (partial-tree (cdr non-left-elts)
                                              right-size)))
              (let ((right-tree (car right-result))
                    (remaining-elts (cdr right-result)))
                (cons (make-tree this-entry left-tree right-tree)
                      remaining-elts))))))))
;a  partial-tree uses a tree recursive style
;b  O(n) growth
;;Exercise 2.65
(define (union-set-1 tree1 tree2)
  (list->tree (union-set (tree->list-2 tree1)
                         (tree->list-2 tree2))))
(define (intersection-set-1 tree1 tree2)
  (list->tree (intersection-set (tree->list-2 tree1)
                                (tree->list-2 tree2))))




;;Exercise 2.73
(define (put-get-gen)
  (define (make-record key1 key2 value)
    (list key1 key2 value))
  (define (key1-record rec) (car rec))
  (define (key2-record rec) (cadr rec))
  (define (value-record rec) (caddr rec))
  (define (match-record? rec key1 key2)
    (and (equal? (key1-record rec) key1)
```

```scheme
        (equal? (key2-record rec) key2)))
  (define table '())
  (list
   (lambda (op type item)
     (define (delete-from tbl key1 key2 result)
       (if (null? tbl)
           result
           (let ((rec (car tbl)))
             (if (match-record? rec key1 key2)
                 (delete-from (cdr tbl) key1 key2 result)
                 (delete-from (cdr tbl) key1 key2 (cons (car tbl) result))
                 ))))
     (let ((new-rec (make-record op type item)))
       (set! table (cons new-rec (delete-from table op type '())))

       new-rec))
   (lambda (op type)
     (define (select-from tbl key1 key2)
       (if (null? tbl)
           #f
           (let ((rec (car tbl)))
             (if (match-record? rec key1 key2)
                 (value-record rec)
                 (select-from (cdr tbl) key1 key2))))
       )
     (select-from table op type))
   (lambda () table)
   ))

(define put-get (put-get-gen))
(define put (car put-get))
(define get (cadr put-get))
(define table (caddr put-get))

(define (install-sum-package)
  (define (addend operands)
    (cadr operands))
  (define (augend operands)
    (let ((rest (cdr operands)))
      (if (null? (cddr rest))
          (cadr rest)
          (cons '+ (cdr rest)))))
  (define (make-sum a1 a2)
    (cond ((=number? a1 0) a2)
```

```
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2)) (+ a1 a2))
        (else (list '+ a1 a2))))
  (define (deriv-sum exp var)
    (make-sum (deriv (addend exp) var)
              (deriv (augend exp) var)))
  (put 'make '+ make-sum)
  (put 'deriv '+ deriv-sum)
  'done
  )

(define (install-product-package)
  (define (multiplier operands)
    (cadr operands))
  (define (multiplicand operands)
    (let ((rest (cdr operands)))
      (if (null? (cddr rest))
          (cadr rest)
          (cons '* (cdr rest)))))
  (define (make-product m1 m2)
    (cond ((or (=number? m1 0) (=number? m2 0)) 0)
          ((=number? m1 1) m2)
          ((=number? m2 1) m1)
          ((and (number? m1) (number? m2)) (* m1 m2))
          (else (list '* m1 m2))))
  (define (deriv-product exp var)
    ((get 'make '+)
     (make-product (multiplier exp)
                   (deriv (multiplicand exp) var))
     (make-product (deriv (multiplier exp) var)
                   (multiplicand exp))))
  (put 'make '* make-product)
  (put 'deriv '* deriv-product)
)

(define (install-exponent-package)
  (define (base operands)
    (cadr operands))
  (define (expt operands)
    (caddr operands))
  (define (make-exponent base expt)
    (cond ((= expt 0) 1)
         ((= expt 1) base)
         (else (list '** base expt))))
```

```scheme
  (define (deriv-expt exp var)
    ((get 'make '*)
     ((get 'make '*) (expt exp)
      (make-exponent (base exp) (- (expt exp) 1)))
     (deriv (base exp) var)))
  (put 'make '** make-exponent)
  (put 'deriv '** deriv-expt)
  'done)

(install-sum-package)
(install-product-package)
(install-exponent-package)
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
        ((get 'deriv '+) exp var))
        ((product? exp)
         ((get 'deriv '*) exp var))
       ((exponentiation? exp)
        ((get 'deriv '**) exp var))
        (else
         (error "unknown expression type -- DERIV" exp))))

(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))

(define (product? x)
  (and (pair? x) (eq? (car x) '*)))

(define (exponentiation? x)
  (and (pair? x) (eq? (car x) '**)))
```