Scheme 2
(define (gcd a b)
  (if (= b 0) a
      (gcd b (remainder a b)))))
Arithmetic operations for rational numbers

- Assume the following functions
  - (make-rat n d) returns a rational number whose numerator is $n$ and whose denominator is $d$
  - (numer x) returns the numerator of a rational number $x$
  - (denom x) returns the denominator of a rational number $x$

(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
             (* (denom x) (denom y))))

(define (-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
             (* (denom x) (denom y))))

(define (*rat x y)
  (make-rat (* (numer x) (numer y))
             (* (denom x) (denom y))))

(define (/rat x y)
  (make-rat (* (numer x) (denom y))
             (* (denom x) (numer y))))

(define (=rat x y)
  (= (* (numer x) (denom y))
      (* (numer y) (denom x))))
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))

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

==> (define one-half (make-rat 1 2))
one-half
(print-rat one-half)
1/2
==> (define one-third (make-rat 1 3))
one-third
==> (print-rat (+rat one-half one-third)
5/6
==> (print-rat (+rat one-third one-third))
6/9

(define (make-rat n d)
  let ((g (gcd n d)))
    cons (/ n g) (/ d g)))
==> (print-rat (+rat one-third one-third))
2/3
A possible way to define \texttt{cons}

\begin{verbatim}
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m)))))
  dispatch)

(define (car z) (z 0))

(define (cdr z) (z 1))
\end{verbatim}
Representing sequences

(cons 1
  (cons 2
    (cons 3
      (cons 4 nil)))))

(list 1 2 3 4)

(define 1-through-4 (list 1 2 3 4))

==> (car 1-through-4)
1

==> (cdr 1-through-4)
(2 3 4)

==> (car (cdr 1-through-4))
2

==> (cons 10 1-through-4)
(10 1 2 3 4)
(define nth n x)
  (if (= n 0)
      (car x)
      (nth (- n 1) (cdr x))))

==> (define squares (list 1 4 9 16 25))
squares
==> (nth 3 squares)
16

(define (length x)
  (if (null? x)
      0
      (+1 (length (cdr x)))))

(define (length x)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter x 0))

==> (define odds (list 1 3 5 7))
odds
==> (append odds squares)
(1 3 5 7 1 4 9 16 25)

(define (append x y)
  (if (null? x)
      y
      (cons (car x) (append (cdr x) y))))
(define (countatoms x)
  (cond (null? x) 0)
    (atom? x) 1)
  (else (+ (countatoms (car x))
             (countatoms (cdr x))))))
• The names CAR and CDR are historical fossils. They relate to assembly language on the IBM 704 computer, the first machine on which Lisp was implemented. The term CAR is short for “Contents of the Address part of Register” and CDR is short for “Contents of the Decrement part of Register”
(define a 1)
a
(define b 2)
b
(list a b)
(1 2)
(list 'a 'b)
(a b)
(list 'a b)
(a 2)
(car '(a b c))
a
(cdr '(a b c))
(b c)

(define (memq item x)
  (cond ((null? x) '())
        ((eq? item (car x)) x)
        (else (memq item (cdr x))))))

(memq 'apple '(pear banana prune))
()
(memq 'apple '(x (apple sauce) y apple pear))
(apple pear)
• Our use of the quotation mark violates the general rule that all compound expressions should be represented as combinations.

• Use `quote` instead:

\[(\text{car } '(a \ b \ c)) = (\text{car } (\text{quote} \ (a \ b \ c)))\]
Example: Symbolic differentiation

(contant? <e>)
(variable? <e>)
(same-variable? <v1> <v2> )
(sum? <e>)
(product? <e>)
(addend <e>)
(augend <e>)
(multiplier <e>)
(multiplicand <e>)
(make-sum <a1> <a2>)
(make-product <m1> <m2>)
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
          (if (same-variable? exp var) 1 0)
        ((sum? exp)
          (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))))))
(define (constant? x) (number? x))
(define (variable? x) (symbol? x))
(define (same-variable? v1 v2) (and (variable? v1) (variable? v2) (eq? v1 v2))
(define (make-sum a1 a2) (list '+ a1 a2))
(define (make-product m1 m2) (list '* m1 m2))
(define (sum? x) (if (not (atom? x)) (eq? (car x) '+) nil))
(define (addend s) (cadr s))
(define (augend s) (caddr s))
(define (product? X) (if (not (atom? X)) (eq? (car x) '* nil))
(define (multiplier p) (cadr p))
(define (multiplicand p) (caddr p))
(define (make-sum a1 a2)
  (cond ((and (number? a1) (number? a2)) (+ a1 a2))
        (number? a1) (if (= a1 0) a2 (list '+ a2 a1))
        (number? a2) (if (= a2 0) a1 (list '+ a1 a2))
        (else (list '+ a1 a2))))
(define (element-of-set? x set)
  (cond ((null? set) nil)
        (((equal? x (car set)) t)
         (else (element-of-set? x (cdr set))))))

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

(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))))
define (element-of-set? x set)
  (cond ((null? set) nil)
        ((= x (car set)) t)
        ((< x (car set)) nil)
        (else (element-of-set? x (cdr set))))

(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2))
      '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2) (cons x1 (intersection-set (cdr set1) (cdr set2))))
              ((< x1 x2) (intersection-set (cdr set1) set2))
              ((< x2 x1) (intersection-set set1 (cdr set2)))))))
Sets - Balanced Trees

(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  (element-of-set? x set)
  (cond ((null? set) nil)
     ((= x (entry set)) t)
     ((< x (entry set)) (element-of-set? x (left-branch set)))
     ((> x (entry set)) (element-of-set? x (right-branch set))))

(define  (adjoin-set x set)
  (cond ((null? set) (make-tree x '() '()))
     ((= x (entry set)) set)
     ((< x (entry set)) (make-tree (entry set)
                               (adjoin-set x (left-branch set))
                               (right-branch set)))
     ((> x (entry set)) (make-tree (entry set)
                               (left-branch set)
                               (adjoin-set x (right-branch set)))))