1 Lists and Trees

(a) You are given two lists:
   (define x (list 1 2 3))
   (define y (list 4 5 6))

What would the following evaluate to:
   (append x y)
   (cons x y)
   (list x y)

(b) Draw the tree structure corresponding to the following list.
   (1 ((2 3) 4) (5 6 7) (8 (9 10)))

(c) Draw the box-and-pointer representation of the above list.
2 sum-leaves

Write a Scheme procedure \texttt{sum-leaves} that takes one argument (a tree with integer leaves) and returns the sum of the leaves.
3 Match

The code on the following pages implements a simple list matcher. The matcher is given two lists as input. The first list (the pattern) contains variables, and the second list (the data) does not. The procedure returns a list of variable bindings that would make the pattern match the data, if possible. Simple variables are denoted by symbols whose first character is a question mark, e.g. ?x.

Evaluate the following expressions:

(match '(a ?x c) '(a b c)) =>

(match '(a (?x c) d) '(a (b c) d)) =>

(match '(a (?x c) ?y) '(a (b c) c)) =>

(match '(a (b c) d) '(a (b c) d)) =>

(match '(a b c d) '(a (b c) c)) =>

(match '(a ?x c ?x e) '(a b c b e)) =>

(match '(a ?x c ?x e) '(a b c d e)) =>

(match '(a ?x c ?y e) '(a b c d e)) =>

(match '(a ?x d) '(a b c d)) =>
Variables are indicated by ?<var>, e.g., ?x.
Binding ?<var>=<value> is represented by (?<var> . <value>)
in an association, e.g., ((?x . a), (?y . b)).
A pattern is a list of symbols and variables.
A datum is a list of symbols.

(define (match p d) (do-match p d *no-bindings*))

;; Arguments: Pattern (p), datum (d), bindings.
;; Returns: A list of bindings or #f

(define (do-match p d bindings)
  (cond ((eq? p d) bindings)
        ((variable? p) (match-variable p d bindings))
        ((segment-variable? p)
         ;; A stand-alone segment variable (match '*x ...) is not well
         ;; defined. The more general case is caught by match-pair.
         *fail*)
        ((pair? p) (match-pair p d bindings))
        (else *fail*)))

;; Arguments: Variable (var), datum (d), bindings.
;; Returns: A list of bindings or #f

(define (match-variable var d bindings)
  (let ((binding (get-binding var bindings)))
    ;; Is the pattern variable on the list of bindings:
    (if binding
      ;; If it is, substitute its value and try again:
      (do-match (binding-val binding) d bindings)
      ;; Otherwise, add new binding:
      (extend-bindings var d bindings))))
Arguments: Pattern list (p), datum (d), bindings.
Returns: A list of bindings or #f

(define (match-pair p d bindings)
  (if (segment-variable? (first p))
      ;; to be implemented in Problem Set 1
      (match-segment-variable p d bindings)
    (if (pair? d) ; p is a pair, so d better be
      (let ((result (do-match (first p) (first d) bindings)))
        ;; See if the FIRST parts match producing new bindings:
        (if (eq? result *fail*)
          ;; If they do not match, fail.
          *fail*
        ;; If they do match, try the REST parts using the
        ;; resulting bindings:
        (do-match (rest p) (rest d) result)
      )))
    *fail*)))

;;; VARIABLE ABSTRACTION

;;; Is x a variable (a symbol beginning with ‘?’)?
(define (variable? x)
  (and (symbol? x) (equal? (string-ref (symbol->string x) 0) #\?)))

;;; Is x a segment variable (a symbol beginning with ‘*’)?
(define (segment-variable? x)
  (and (symbol? x) (equal? (string-ref (symbol->string x) 0) #\*)))

;;; BINDINGS ABSTRACTION

;;; Indicates unification/match failure
(define *fail* #f)

;;; Indicates unification/match success, with no variables.
(define *no-bindings* '(()f))
;;; Find a (variable . value) pair in a binding list.
(define (get-binding var bindings)
    (assoc var bindings))

;;; Get the variable part of a single binding.
(define (binding-var binding)
    (and binding (car binding)))

;;; Get the value part of a single binding.
(define (binding-val binding)
    (and binding (cdr binding)))

(define make-binding cons)

;;; Add a (var . value) pair to a binding list.
(define (extend-bindings var val bindings)
    (cons (make-binding var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy *no-bindings*
        (if (eq? bindings *no-bindings*)
            ()
            bindings)))

;;;; HELPER FUNCTION

;;;; Add elt to the end of lst.
(define (add-to-end elt lst)
    (append lst (list elt)))

(define rest cdr)
4 Scheme Quick Reference

Scheme specification: http://www.schemers.org/Documents/Standards/R5RS/

Notation:
<object> any Scheme data object.
<object>* zero or more objects
<object>+ one or more objects
<object>[] optional object
( <whatever> )... Zero or more occurrences of ( <whatever> )

Built-in Commands Signatures:
(lambda <name> <exp>+ )
(lambda (<name>* ) <exp>+ )
(and <exp>* )
(or <exp>* )
(if <test-exp> <if-true> [<if-false>] )
(cond (<test> <exp>* )... [(else <exp>+ )] )
(case <key-exp> ((<datum>+ ) <exp>* )... [(else <exp>+ )] )
(define ( <name> <name>* ) <exp>+ )
(define <name> <exp> )
(let [<name>] ( (vname<value-exp>)... ) <exp>+ )
(let* ( (vname<value-exp>)... ) <exp>+ )
(letrec ( (vname<value-exp>)... ) <exp>+ )
(begin <expression>+ )
(do ( (var<init><step>)... ) (test<exp>* ) <exp>* )

(not <object>)
(boolean? <object>)
(eq?  <obj1>  <obj2>)
(eqv?  <obj1>  <obj2>)
(equal?  <obj1>  <obj2>)

(pair?  <object>)
(cons  <obj1>  <obj2>)
(car  <pair>)
(cdr  <pair>)
(set-car!  <pair>  <object>)
(set-cdr!  <pair>  <object>)

(caar  <list>)  (cadr  <list>)  (cdar  <list>)  (cddr  <list>)
(caaar  <list>)  (caadr  <list>)  (cadar  <list>)  (caaddr  <list>)
(cdaar  <list>)  (cdaadr  <list>)  (cdadar  <list>)  (cddadr  <list>)
(cdaaar  <list>)  (cdaadr  <list>)  (cdadar  <list>)  (cddadr  <list>)
(cddaar  <list>)  (cddadr  <list>)  (cdddar  <list>)  (cddddr  <list>)
(null?  <object>)
(list?  <object>)
(list  <object>* )
(length  <list>)
(append  <list>+ )
(reverse  <list>)
(list-ref  <list>  <index>)

(memq  <object>  <list>)
(memv  <object>  <list>)
(member  <object>  <list>)

(assq  <object>  <alist>)
(assv  <object>  <alist>)
(assoc  <object>  <alist>)

(symbol?  <object>)  (symbol->string  <symbol>)  (string->symbol  <string>)

(number?  <object>)
(complex?  <object>)
(real?  <object>)
(rational?  <object>)
(integer?  <object>)
(exact?  <number>)  (inexact?  <number>)
(=  <number>+  )
(<  <number>+  )  (>=  <number>+  )
(positive?  <number>)  (negative?  <number>)
(odd?  <number>)  (even?  <number>)
(max  <number>+  )  (min  <number>+  )
(+  <number>+  )
(*  <number>+  )
(-  <number>+  )
(/  <number>+  )
(abs  <number>)
(quotient  <num1>  <num2>)  (remainder  <num1>  <num2>)
(modulo  <num1>  <num2>)
(gcd  <number>*  )  (lcm  <number>*  )
(numerator  <rational>)  (denominator  <rational>)
(floor  <number>)  (ceiling  <number>)
(truncate  <number>)  (round  <number>)
(rationalize  <num1>  <num2>)
(exp  <number>)  (log  <number>)
(sin  <number>)  (cos  <number>)  (tan  <number>)
(asin  <number>)  (acos  <number>)  (atan  <number> [<number>])
(sqrt  <number>)
(expt  <num1>  <num2>)
(make-rectangular  <num1>  <num2>)  (make-polar  <num1>  <num2>)
(real-part  <number>)  (imag-part  <number>)
(magnitude  <number>)  (angle  <number>)
(exact->inexact  <number>)  (inexact->exact  <number>)
(number->string  <number>)  (string->number  <string>)

(char?  <object>)
(char=?  <char1>  <char2>)  (char-ci=?  <char1>  <char2>)
(char?  <char1>  <char2>)  (char-ci<?  <char1>  <char2>)
(char=?  <char1>  <char2>)  (char-ci?>  <char1>  <char2>)
(char=?  <char1>  <char2>)  (char-ci<=?  <char1>  <char2>)
(char=?  <char1>  <char2>)  (char-ci=>?  <char1>  <char2>)
(char-alphabetic?  <character>)
(char-numeric?  <character>)

9
(char-whitespace? <character>)
(char-upper-case? <character>) (char-lower-case? <character>)
(char->integer <character>) (integer->char <integer>)
(char-upcase <character>) (char-downcase <character>)

(string? <object>)
(make-string <length> [<character>] )
(string <character>+ )
(string-length <string>)
(string-ref <string> <index>)
(string-set! <string> <index> <character>)

(string=? <string1> <string2>) (string-ci=? <string1> <string2>)
(string<? <string1> <string2>) (string-ci<? <string1> <string2>)
(string>? <string1> <string2>) (string-ci>? <string1> <string2>)
(string<=? <string1> <string2>) (string-ci<=? <string1> <string2>)
(string=>? <string1> <string2>) (string-ci=>? <string1> <string2>)
(substring <string> start-index end-index)
(string-append <string>+ )

(vector? <object>)
(make-vector <length> [<object>] )
(vector <object>* )
(vector-length <vector>)
(vector-ref <vector> <index>)
(vector-set! <vector> <index> <object>)

(procedure? <object>)
(apply <procedure> <arg>* <arg-list>)
(map <procedure> <list>+ )
(for-each <procedure> <list>+ )
(call-with-current-continuation <one-argument-procedure>)

(call-with-input-file <string> <procedure>)
(call-with-output-file <string> <procedure>)
(input-port? <object>) (output-port? <object>)
(current-input-port) (current-output-port)
(open-input-file <string>) (open-output-file <string>)
(close-input-port <input-port>) (close-output-port <output-port>)
(eof-object? <object>)
(read [<input-port>] )
(read-char [<input-port>] )
(peek-char [<input-port>] )
(write  <object> [<output-port>] )
(display <object> [<output-port>] )
(newline [<output-port>] )
(write-char <character> [<output-port>] )

(list-tail <list> <index>)
(string->list <string>)
(list->string <list-of-characters>)
(string-copy <string>)
(string-fill! <string> <character>)
(vector->list <vector>)
(list->vector <list>)
(vector-fill! <vector> <object>)
(delay <expression>)
(force <promise>)
(with-input-from-file <string> <thunk>)
(with-output-to-file <string> <thunk>)
(char-ready? [<input-port>] )
(load <string>)
(transcript-on <string>)
(transcript-off)