lib/basic/list.ath

```
(define (for-each L pred?)
  (match L
    ([] true)
    ((list-of x xs) (&& (pred? x) (for-each xs pred?)))))

(define (for-some L pred?)
  (match L
    ([] false)
    ((list-of x xs) (|| (pred? x) (for-some xs pred?)))))

(define (negate t)
  (match t
    (true false)
    (false true)))

(define (for-none L pred?)
  (for-each L (lambda (x) (negate (pred? x)))))

(define (member? x L)
  (for-some L (lambda (y) (equal? x y))))

# The following returns the index of the first (leftmost) occurrence of x in L.
# If x does not occur in L, then false is returned. Index-counting starts with 1.

(define (member-index x L)
  (letrec ((search (lambda (L i)
                     (match L
                       ([] false)
                       ((list-of y rest) (check ((equal? x y) i)
                                                (else (search rest (plus i 1)))))))))
    (search L 1)))

(define (member-eq? x L eq?)
  (match L
    ([] false)
    ((list-of y rest) (check ((eq? x y) true)
                             (else (member-eq? x rest eq?))))))

(define (subset? L1 L2)
  (for-each L1 (lambda (x) (member? x L2))))

(define (subset-eq? L1 L2 eq)
  (for-each L1 (lambda (x) (member-eq? x L2 eq))))

(define (equal-lists-as-sets L1 L2)
  (&& (subset? L1 L2)
      (subset? L2 L1)))

(define (for-some' list pred?)
  (letrec ((loop (lambda (previous rest)
                   (match rest
                     ([] [])
                     ((list-of x more) (check ((pred? x) [(rev previous) x more])
                                              (else (loop (add x previous) more)))))))
     (loop [] list)))

(define (filter L f)
  (letrec ((loop (lambda (L results)
                  (match L
                   ([] (rev results))
                   ((list-of x rest) (check ((f x) (loop rest (add x results)))
                                            (else (loop rest results))))))))
    (loop L [])))

(define (filter-out L f)
  (filter L (lambda (x) (negate (f x)))))

(define (filter-and-complement L f)
```

```
  (letrec ((loop (lambda (L sat unsat)
                   (match L
                     ([] [sat unsat])
                     ((list-of x rest) (check ((f x) (loop rest (add x sat) unsat))
                                              (else  (loop rest sat (add x unsat)))))))))
    (loop L [] [])))

(define (filter-and-complement-rev L f)
  (letrec ((loop (lambda (L sat unsat)
                   (match L
                     ([] [(rev sat) (rev unsat)])
                     ((list-of x rest) (check ((f x) (loop rest (add x sat) unsat))
                                              (else  (loop rest sat (add x unsat)))))))))
    (loop L [] [])))

(define (map-proc f l)
  (match l
    ([] ())
    ((list-of x rest) (seq (f x) (map-proc f rest)))))

(define iter map-proc)

(define (map-proc2 f l)
  (letrec ((loop (lambda (L)
                   (match L
                     ([] ())
                     ((list-of x rest) (seq (f x) (loop rest)))))))
     (loop l)))

(define (list-diff L1 L2)
  (let ((len1 (length L1))
        (len2 (length L2)))
    (check ((greater? (plus len1 len2) 100)
              (let ((T (table len2))
                    (_ (map-proc (lambda (x) (table-add T [x --> true])) L2))
                    (in-L2 (lambda (x) (try (table-lookup T x) false))))
               (filter L1 (lambda (x) (negate (in-L2 x))))))
           (else (filter L1 (lambda (x) (negate (member? x L2))))))))

(define (remove x L)
  (letrec ((patch (lambda (front back)
                    (match front
                      ([] back)
                      ((list-of y rest) (patch rest (add y back))))))
           (loop (lambda (L front)
                   (match L
                     ((list-of (val-of x) rest) (patch front (loop rest [])))
                     ((list-of y rest) (loop rest (add y front)))
                     (_ front)))))
    (loop L [])))


(define (remove x L)
  (letrec ((patch (lambda (front back)
                    (match front
                      ([] back)
                      ((list-of y rest) (patch rest (add y back))))))
           (loop (lambda (L front)
                   (match L
                     ((list-of (val-of x) rest) (patch (rev front) (loop rest [])))
                     ((list-of y rest) (loop rest (add y front)))
                     (_ (rev front))))))
    (loop L [])))

(define (list-remove x L) (remove x L))

(define (remove-and-preserve-order a L)
  (letrec ((loop (lambda (rest already-seen)
                   (match rest
                     ((list-of x more)
```

```
                         (check ((equal? x a) (loop more already-seen))
                                (else (loop more (add x already-seen)))))
                      (_ (rev already-seen))))))
    (loop L [])))

(define list-remove remove-and-preserve-order)

(define (rd L)
  (letrec ((loop (lambda (L res)
                    (match L
                      ((list-of x more) (check ((member? x res) (loop more res))
                                               (else (loop more (add x res)))))
                      ([] res)))))
    (loop L [])))

(define (rd-eq L equal?)
  (letrec ((loop (lambda (L res)
                    (match L
                      ((list-of x more) (check ((member-eq? x res equal?) (loop more res))
                                               (else (loop more (add x res)))))
                      ([] res)))))
    (loop L [])))

(define (remove-duplicates L)
  (let ((T (table 50))
        (occurs-once (lambda (x)
                         (try (table-lookup T x)
                              (seq (table-add T [x false])
                                   true)))))
    (filter L occurs-once)))

(define rd remove-duplicates)

(define remove-duplicates-eq rd-eq)

(define (map-with-index f L)
  (letrec ((loop (lambda (L results i)
                    (match L
                      ([] (rev results))
                      ((list-of x rest) (loop rest (add (f x i) results) (plus i 1)))))))
    (loop L [] 1)))

(define (app-with-index f L)
  (letrec ((loop (lambda (L i)
                    (match L
                      ([] ())
                      ((list-of x rest) (seq (f x i) (loop rest (plus i 1))))))))
    (loop L 1)))

(define (map2 f L1 L2)
  (letrec ((loop (lambda (L1 L2 results)
                    (match [L1 L2]
                      ([[] _] (rev results))
                      ([_ []] (rev results))
                      ([(list-of x xs) (list-of y ys)]
                       (loop xs ys (add (f x y) results)))))))
    (loop L1 L2 [])))

# (map2 add [1 2 3] [[4 5] [6 7] [8 9 10]]) = [[1 4 5] [2 6 7] [3 8 9 10]]
# (map2 add [1] [[4 5] [6 7] [8 9 10]])     = [[1 4 5]]
# (map2 add [] [[4 5] [6 7] [8 9 10]])      = []
# (map2 add [1 2 3] [])                     = []
# To test the above:
# (let ((a 'a) (b 'b) (c 'c) (d 'd) (e 'e) (f 'f))
#   (seq
#     (writeln-val (map2 add [1 2 3] [[4 5] [6 7] [8 9 10]]))
#     (writeln-val (map2 add [1] [[4 5] [6 7] [8 9 10]]))
#     (writeln-val (map2 add [] [[4 5] [6 7] [8 9 10]]))
#     (writeln-val (map2 add [1 2 3] []))))
```

```
define (non-empty? L) := (match L ((list-of _ _) true) ([] false))

(define (map-select f L pred)
  (letrec ((loop (lambda (L results)
                   (match L
                     ([] (rev results))
                     ((list-of x rest) (let ((res (f x)))
                                         (check ((pred res) (loop rest (add res results)))
                                                (else (loop rest results)))))))))
     (loop L [])))

(define (map-select-2 f L pred)
  (letrec ((loop (lambda (L results)
                   (match L
                     ([] (rev results))
                     ((list-of x rest) (check ((pred x) (loop rest (add (f x) results)))
                                              (else (loop rest results))))))))
     (loop L [])))

(define map-list map)

(define (zip L1 L2)
  (letrec ((f (lambda (L1 L2 res)
                (match [L1 L2]
                  ([(list-of x1 rest1) (list-of x2 rest2)] (f rest1 rest2 (add [x1 x2] res)))
                  (_ (rev res))))))
    (f L1 L2 [])))

(define list-zip zip)

(define (unzip L)
  (letrec ((loop (lambda (L L1 L2)
                   (match L
                     ([] [(rev L1) (rev L2)])
                     ((list-of [x y] rest) (loop rest (add x L1) (add y L2)))))))
    (loop L [] [])))

## Here the input method M is a regular method
## that takes an input x (usually a sentence) and produces
## a theorem. We apply M to all elements of the input list L,
## collecting the results along the way, ultimately passing them
## to the continuation K. Note that an error will result if
## the application of M to an element of the list L fails.

(define (map-method M L K)
  (dletrec ((loop (method (L res)
                    (dmatch L
                      ([] (!K (rev res)))
                      ((list-of x rest) (dlet ((th (!M x)))
                                          (!loop rest (add th res))))))))
     (!loop L [])))

## Here the input method M is also a regular method
## that takes an input x (usually a sentence) and produces
## a theorem, but we don't fail if applying M to an
## element of the list L produces an error. Conceivably,
## M could fail on every element of L, in which case we
## will simply pass the empty list to the continuation K:

(define (map-method-non-strictly M L K)
  (dletrec ((loop (method (L res)
                    (dmatch L
                      ([] (!K (rev res)))
                      ((list-of x rest) (dlet ((ok? (cell false))
                                               (th (dtry (dlet ((res (!M x))
                                                                (_ (set! ok? true)))
                                                           (!claim res))
                                                         (!true-intro)))
                                               (res' (check ((ref ok?) (add th res))
                                                            (else res))))
```

```
                                          (!loop rest res')))))))
     (!loop L []))))

(define (map-method-non-strictly-2 M L K)
  (dletrec ((loop (method (L res productive)
                    (dmatch L
                      ([] (!K (rev res) productive))
                      ((list-of x rest) (dlet ((ok? (cell false))
                                               (th (dtry (dlet ((res (!M x))
                                                                (_ (set! ok? true)))
                                                           (!claim res))
                                                         (!true-intro)))
                                               ([res' productive'] (check ((ref ok?) [(add th res) (add x productive)]
                                                                          (else [res productive]))))
                                          (!loop rest res' productive')))))))
     (!loop L [] []))))

## map-methods-non-strictly takes a list of methods [M_1 ... M_k] and (non-strictly) applies
## each M_i to the input list L, keeping track of the results along the way. All results
## thereby obtained are ultimately passed to the continuation K:

(define (map-methods-non-strictly method-list L K)
  (dletrec ((loop (method (methods all-results)
                    (dmatch methods
                      ([] (!K all-results))
                      ((list-of M rest) (!map-method-non-strictly M L
                                           (method (new-results)
                                             (!loop rest (join new-results all-results)))))))))
     (!loop method-list []))))



(define (map-methods-non-strictly-2 method-list L K)
  (dletrec ((loop (method (methods all-results productive)
                    (dmatch methods
                      ([] (!K all-results (remove-duplicates productive)))
                      ((list-of M rest) (!map-method-non-strictly-2 M L
                                           (method (new-results new-productive)
                                             (!loop rest (join new-results all-results)
                                                         (join new-productive productive)))))))))
     (!loop method-list [] []))))

## map-methods-non-strictly* is similar, but iterates this process up to a fixed
## point, or until the number of iterations exceeds the limit specified by max-iterations,
## whichever occurs first. (The parameter max-iterations is optional. The unit value,
## or any other non-numeric value can be given to it, in which case the method will
## iterate up to a fixed point.) Specifically, first we apply all methods to the input
## list L, obtaining the first generation of results. Then we apply the methods to
## those results, obtaining a second generation of results, and so on. A fixed point
## is reached when the latest yield of results does not give us anything we haven't
## already obtained in some previous generation. Finally, all the results are
## cleaned up (removing duplicates, etc.) and passed to the continuation K:

(define (map-methods-non-strictly* method-list L K max-iterations)
  (dlet ((test (match max-iterations
                 (x:Int (lambda (j) (greater? j max-iterations)))
                 (_ (lambda (_) false)))))
    (dletrec ((loop (method (latest-results all-results i)
                      (!map-methods-non-strictly method-list latest-results
                         (method (new-results)
                           (dcheck ((|| (test i) (subset? new-results all-results))
                                     (!K (list-diff (remove-duplicates all-results) L)))
                                   (else (!loop new-results (join new-results all-results) (plus i 1)))))))))
       (!map-methods-non-strictly method-list L
         (method (first-results)
           (!loop first-results first-results 1))))))

(define (map-methods-non-strictly-2* method-list L K max-iterations)
  (dlet ((test (match max-iterations
                 (x:Int (lambda (j) (greater? j max-iterations)))
```

```
                  (_ (lambda (_) false)))))
    (dletrec ((loop (method (latest-results all-results productive i)
                      (!map-methods-non-strictly-2 method-list latest-results
                         (method (new-results new-productive)
                           (dcheck ((|| (test i) (subset? new-results all-results))
                                      (!K (list-diff (remove-duplicates all-results) (join L productive))))
                                    (else (!loop new-results (join new-results all-results)
                                                             (join new-productive productive) (plus i 1)))))))))
       (!map-methods-non-strictly-2 method-list L
         (method (first-results productive)
           (!loop first-results first-results productive 1))))))

##
## A "multi-method" is one that takes an input x and produces a *list*
## of theorems. For that reason,  a multi-method must be binary: it takes
## a continuation K as a second argument, and passes the list of its results
## to it when it is done. A typical example of a multi-method is a conjunction
## elimination method that returns both the left and the right component
## of a given conjunction:
##
##
## (define (conj-elim p K)
##   (dmatch p
##      ((and _ _) (dlet ((p1 (!left-and p))
##                        (p2 (!right-and p)))
##                   (!K [p1 p2])))
##      (_ (!K [])))))
##
## Note that multi-methods should be written to be fail-safe, so that
## if an error occurs, the empty list of theorems is passed to the given
## continuation (instead of halting execution with an error). This is
## the reason why conj-elim was written as above with pattern matching,
## rather than as follows:
##
## (define (conj-elim p K)
##   (dlet ((p1 (!left-and p))
##          (p2 (!right-and p)))
##     (!K [p1 p2])))
##
## Of course, the first way of writing the method is not 100% fail-safe
## either. E.g., if it's applied to a conjunction that is not in the a.b.,
## we'll get an error. We can make a multi-method completely safe
## as follows:
##
## (define (make-multi-method-safe M)
##   (method (L K)
##     (dtry (!M L K)
##           (!K [])))))
##
## However, use of make-multi-method-safe is inefficient and not recommended.
##
## "map-multi-method" below must be used to map a multi-method M to a list:
##

(define (map-multi-method M L K)
  (dletrec ((loop (method (L results)
                    (dmatch L
                      ([] (!K (rev results)))
                      ((list-of x rest) (!M x (method (new-results)
                                                (!loop rest (join new-results results)))))))))
    (!loop L [])))

## A list method is one that can be applied to a list of inputs
## and return a list of theorems. Thus a list method must be binary,
## taking a continuation K as its second argument. The list of resulting
## theorems is eventually passed to the continuation. Any regular method
## (taking an input x and producing a theorem) can be turned into
## a list method as follows:

(define (make-list-method M)
```

```
  (method (L K)
    (!map-method-non-strictly M L K)))


(define (map-multi-method* M L K max)
  (dletrec ((loop (method (latest-results all-results i)
                   (!map-multi-method M latest-results
                      (method (new-results)
                        (dcheck ((|| (subset? new-results all-results)
                                     (less? max i)
                                     (equal? max i))
                                 (!K (remove-duplicates all-results)))
                               (else (!loop new-results (join new-results all-results) (plus i 1)))))))))
    (!map-multi-method M L
      (method (results)
        (!loop results results 1)))))



(define (fold f lst id)
  (match lst
    ([] id)
    ([x] (f x id))
    ((list-of x (list-of y more))
       (fold f (add (f x y) more) id))))

(define (foldl f e l)
  (match l
    ([] e)
    ((list-of x xs) (foldl f (f e x) xs))))

(define (foldr f e l)
  (match l
    ([] e)
    ((list-of x rest) (f x (foldr f e rest)))))

define (flatten L) := (foldl join [] L)


(define (nth i l)
  (match [i l]
    ([1 (list-of x _)] x)
    (_ (nth (minus i 1) (tail l)))))

## The above gets into an infinite loop for negative indices.
## ith below fixes that. It also takes the list as the first
## argument, the index

(define (ith L i)
  (check ((greater? i 1) (ith (tail L) (minus i 1)))
         ((equal? i 1) (head L))))

(define at ith)

(set-precedence at 107)

(define (nth i L) (ith L i))

(define (nth-tail l n)
  (match n
    (0 l)
    (_ (nth-tail (tail l) (minus n 1)))))

(define (decompose-nth L n)
  (letrec ((loop (lambda (L L' i ith)
                  (match [i L]
                    ([1 (list-of x rest)] (loop rest L' (minus i 1) x))
                    ([_ (list-of x rest)] (loop rest (add x L') (minus i 1) ith))
                    ([_ []] [ith L'])))))
    (loop L [] n ())))
```

```
(define (prefix? s1 s2)
  (match [s1 s2]
    ([[] _] true)
    ([(list-of x rest1) (list-of x rest2)] (prefix? rest1 rest2))
    (_ false)))

(define (find-min-rest l fun)
  (letrec ((f (lambda (rem rest min i)
                (match rem
                  ([] [min rest])
                  ((list-of x more) (check
                                      ((greater? i 0) (check ((less? (fun x) (fun min)) (f more (add min rest)
                                                                 x (plus i 1)))
                                                             (else (f more (add x rest) min (plus i 1)))))
                                      (else (f more [] x (plus i 1)))))))))
    (check ((null? l) ()) (else (f l [] () 0)))))

(define (non-null L) (negate (null? L)))

(define (take l n)
  (letrec ((f (lambda (l n res)
                (match [l n]
                  ([_ 0]  (rev res))
                  ([[] _] (rev res))
                  (_ (f (tail l) (minus n 1) (add (head l) res)))))))
    (f l n [])))

(define (drop L n)
  (check ((less? n 1) L)
         (else (drop (tail L) (minus n 1)))))

(define (split-list l n)
  (letrec ((f (lambda (l n res)
                (match [l n]
                  ([_ 0]  [(rev res) l])
                  ([[] _] [(rev res) l])
                  (_ (f (tail l) (minus n 1) (add (head l) res)))))))
    (f l n [])))

(define (cut-in-half L)
  (split-list L (div (length L) 2)))

(define (from-to i j)
  (letrec ((loop (lambda (k res)
                   (check ((less? k i) res)
                          (else (loop (minus k 1) (add k res)))))))
     (check ((less? j i) [])
            (else (loop j [])))))

(define to from-to)

(define (quant* Q var-list P)
 (match var-list
   ([] P)
   ((list-of v more-vars) (Q v (quant* Q more-vars P)))))

(define (forall* vl P) (quant* forall vl P))

(define (exists* vl P) (quant* exists vl P))

(define (close x)
  (match x
    ((some-sentence p) (forall* (free-vars p) p))
    ((some-list L) (map close L))
    (_ x)))

(define (make-selector-axioms S)
  (let ((constructors (constructors-of S))
        (selectors-of (lambda (con) (map-select-2 string->symbol
```

```
                                                    (selector-names con)
                                                    (lambda (str) (negate (null? str))))))
        (make-straight-axioms
           (lambda (con)
             (let ((range (from-to 1 (arity-of con)))
                   (vars (map (lambda (_) (fresh-var)) range))
                   (con-term (make-term con vars))
                   (sel-vars (zip (selectors-of con) vars)))
               (map (lambda (sel-var-pair)
                      (match sel-var-pair
                        ([sel v] (close (= (sel con-term) v)))))
                    sel-vars))))
       (make-option-axioms
           (lambda (con)
             (let ((range (from-to 1 (arity-of con)))
                   (vars (map (lambda (_) (fresh-var)) range))
                   (con-term (make-term con vars))
                   (selectors (selectors-of con))
                   (sel-vars (zip selectors vars))
                   (other-constructors (filter constructors (lambda (c) (negate (equal? c con)))))
                   (main-axioms (map (lambda (sel-var-pair)
                                        (match sel-var-pair
                                         ([sel v] (close (= (sel con-term) ((string->symbol "SOME") v))))))
                                      sel-vars))
                   (option-axioms (lambda (sel)
                                      (map (lambda (c)
                                              (let ((con-term (make-term c (map (lambda (_) (fresh-var)) (from-to 1 (ar
                                                (close (= (sel con-term) (string->symbol "NONE")))))
                                           other-constructors)))
                  (all-option-axioms (flatten (map option-axioms selectors))))
              (join main-axioms all-option-axioms)))))
   (match (struc-with-option-valued-selectors? S)
     (true (flatten (map make-option-axioms constructors)))
     (_ (flatten (map make-straight-axioms constructors))))))


(define selector-axioms make-selector-axioms)

(define (upper-case-alpha-char? c)
  (&& (member? (compare-chars c `A) ['greater 'equal])
      (member? (compare-chars c `Z) ['less 'equal])))

(define (downcase c)
  (check ((upper-case-alpha-char? c) (char (plus (char-ord c) 32)))
         (else c)))

(define (downcase-string str)
  (map downcase str))

(define (separate L token)
  (match L
    ([] "")
    ([s] s)
    ((list-of s1 (bind rest (list-of _ _)))
      (join s1 token (separate rest token)))))

(define (auto-induction-definition structure-name)
  (let ((prequel (join "(define (" (downcase-string structure-name) "-induction goal-property)\n"))
        (make-constructor-line
          (lambda (c)
             (let ((arity (arity-of c))
                   (pattern (check ((less? arity 1)
                                      (join "(bind state " (symbol->string c) ")"))
                                   (else (join "(bind state (" (symbol->string c) " " (separate (map (lambda (_) "_")
                   (body (join "(!prove (goal-property state))")))
               (join "(" pattern " " body ")"))))
       (lines (separate (map make-constructor-line (constructors-of structure-name)) "\n"))
       (cmd (join prequel "\n (by-induction (forall ?x (goal-property ?x))\n" lines "))")))
    (process-input-from-string cmd)))
```

```
(make-private "auto-induction-definition")

# New additions, July 05 2015:

# (datatype (List-Of T)
#   Nil
#   (Cons hd:T tl:(List-Of T)))

#datatype (List T) := nil | (cons hd:T tl:(List T))

datatype (List T) := nil | (:: hd:T tl:(List T))

define [Nil Cons] := try { [nil ::] | [nil cons] }

 (define (makeList L)
  (match L
   ([] Nil)
   ((list-of x rest) (Cons x (makeList rest)))))


(define (alist->list L)
  (match L
    ([] (Nil))
    ((list-of x rest) (Cons x (alist->list rest)))
    (_ L)))


(define (alist->list inner)
  (letrec ((loop (lambda (L)
                   (match L
                     ([] (Nil))
                     ((list-of x rest) (Cons (inner x) (loop rest)))
                     (_ L)))))
     loop))

(define (list->alist L)
  (match L
    (Nil [])
    ((Cons x rest) (add x (list->alist L)))
    (_ L)))

(define (list->alist inner)
  (letrec ((loop (lambda (L)
                   (match L
                     (Nil [])
                     ((Cons x rest) (add (inner x) (loop rest)))
                     (_ L)))))
     loop))


(define
  (even-positions L)
    (match L
      ((list-of _ rest) (odd-positions rest))
      (_ []))
  (odd-positions L)
    (match L
      ((list-of x rest) (add x (even-positions rest)))
      (_ []))))

(define (merge L1 L2 less?)
  (match [L1 L2]
    ([(list-of x rest1) (list-of y rest2)]
        (check ((less? x y) (add x (merge rest1 L2 less?)))
               (else (add y (merge rest2 L1 less?)))))
    ([[] _] L2)
    (_ L1)))
```

```
(define (merge-sort' L less?)
  (match L
    ((list-of _ (list-of _ rest)) (merge (merge-sort' (odd-positions L) less?)
                                         (merge-sort' (even-positions L) less?) less?))
    (_ L)))

(define (merge-sort L less?)
   (letrec ((sort (lambda (L)
                    (match L
                     ((list-of _ (list-of _ _))
                        (let (([L1 L2] (cut-in-half L)))
                          (merge (sort L1) (sort L2) less?)))
                     (_ L)))))
    (sort L)))

(define (skip-until list pred)
  (match list
    ([] [])
    ((list-of x rest) (check ((pred x) list)
                             (else (skip-until rest pred))))))


(define (skip-until' list pred)
  (letrec ((loop (lambda (L so-far)
                   (match L
                      ([] [(rev so-far) []])
                      ((list-of c more)
                          (check ((pred c) [(rev so-far) L])
                                 (else (loop more (add c so-far)))))))))
    (loop list [])))



(define (tokenize L delims)
  (let ((pred (lambda (c)
                (member? c delims)))
        (pred' (lambda (c) (negate (member? c delims))))
        (add' (lambda (str tokens)
                 (check ((null? str) tokens)
                        (else (add str tokens))))))
    (letrec ((loop (lambda (L tokens)
                      (match L
                         ([] (rev tokens))
                         (_ (let (([_ rest] (skip-until' L pred'))
                                  ([token more] (skip-until' rest pred)))
                              (loop more (add' token tokens))))))))
      (loop L []))))

(define tokenize-string tokenize)

# (first-image-that-works L f) returns (f x) for the first (leftmost) element x
# of L such that (f x) is defined (its computation doesn't lead to an error).
# If L has no such x, (first-image-that-works L f) fails in error:

define (first-image L f) :=
  match L {
    (list-of x rest) => try { (f x) | (first-image rest f) }
  }

(define (find-element L pred success failure)
  (letrec ((search (lambda (list)
                     (match list
                       ([] (failure))
                       ((list-of x rest) (check ((pred x) (success x))
                                                (else (search rest))))))))
     (search L)))

##  (find-element' L pred f success failure) finds the first element
##  x of L such that (pred (f x)) holds, if such x exists in L.
```

```
##  If so, then (success (f x)) is called, otherwise (failure) is called.

(define (find-element' L pred f success failure)
  (letrec ((search (lambda (list)
                     (match list
                       ([] (failure))
                       ((list-of x rest)
                          (let ((res (f x)))
                            (check ((pred res) (success res))
                                   (else (search rest)))))))))
     (search L)))

(define find-some-element find-element')

## (find-first L f) finds the first (leftmost) member x of list L such that
## (f x) returns something - call it y - other than false. That result, y,
## is the result of (find-first L pred). If L contains no such x, then an error occurs:

(define (find-first L f)
  (match L
    ((list-of x rest) (match (f x)
                        (false (find-first rest f))
                        (res res)))))

# find-first' works like find-first, except that if there is no element x in L
# such that (f x) is non-false, then the nullary continuation K is invoked:

(define (find-first' L f K)
  (match L
    ((list-of x rest) (match (f x)
                        (false (find-first' rest f K))
                        (res res)))
    (_ (K))))



(define (find-list-member L pred? success failure)
  (dmatch L
    ([] (!failure))
    ((list-of x rest) (dcheck ((pred? x) (!success x))
                              (else (!find-list-member rest pred? success failure))))))


(define (find-some L M K)
  (dmatch L
    ([] (!K))
    ((list-of x rest) (dtry (!M x)
                            (!find-some rest M K)))))


# (find-in-list L P) takes finds the first element y in L such
# that y has the unary property P. In addition, the lists to the
# left and right of y (in L) are also returned. E.g.,
# (find-in-list [0 6 3 4 10] odd?) returns [[0 6] 3 [4 10]].
# If no such y is found, the unit () is returned.

(define (find-in-list L P)
  (letrec ((loop (lambda (L prefix)
                   (match L
                     ([] ())
                     ((list-of x more) (check ((P x) [(rev prefix) x more])
                                              (else (loop more (add x prefix)))))))))
    (loop L [])))

(define (find-decomposition L P)
  (letrec ((loop (lambda (L prefix)
                   (match L
                     ([] ())
                     ((list-of x more) (check ((P prefix x more) [prefix x more])
                                              (else (loop more (join prefix [x]))))))))))
```

```
    (loop L [])))


(define (max i j)
  (check ((less? i j) j)
         (else i)))

(define (min i j)
  (check ((less? i j) i)
         (else j)))

(define (min-or-max L seed f)
  (match L
    ([] seed)
    ((list-of x rest) (min-or-max rest (f seed x) f))))

(define (generic-list-min L generic-min)
  (match L
    ([] ())
    (_  (min-or-max L (head L) generic-min))))

(define (list-max L)
  (match L
    ([] ())
    (_ (min-or-max L (head L) max))))

(define (list-min L)
  (match L
    ([] ())
    (_ (min-or-max L (head L) min))))


(define (max* L)
  (match L
    ([] 0)
    (_ (list-max L))))

(define (intersection A B)
  (letrec ((loop (lambda (A B result)
                   (match A
                     ([] result)
                     ((list-of x Amore)
                      (check ((&& (member? x B) (negate (member? x result)))
                              (loop Amore B (add x result)))
                             (else (loop Amore B result))))))))
    (loop A B [])))


# define (fast-intersection A B) :=
#   let {T := (list->table A)}
#    (filter B
#            lambda (x) (x HashTable.in T))

# (define (fast-intersection* lists)
#   (match lists
#     ([] [])
#     ([L] L)
#     ((list-of L more)
#      (fast-intersection L (fast-intersection* more)))))

(define list-intersection intersection)

(define (intersection* lists)
  (match lists
    ([] [])
    ([L] L)
    ((list-of L more)
     (intersection L (intersection* more)))))

(define (fast-intersection L1 L2)
```

```
  (let ((T (table 10))
        (_ (map-proc (lambda (x) (table-add T [x --> true])) L1)))
    (filter L2 (lambda (x) (try (table-lookup T x) false)))))

(define (all-but-last L)
  (match L
    ([] [])
    ([x] [])
    ((list-of x rest) (add x (all-but-last rest)))))

(define (find L pred success-cont failure-cont)
  (letrec ((loop (lambda (L)
                   (match L
                     ([] (failure-cont))
                     ((list-of x rest) (check ((pred x) (success-cont x))
                                              (else (loop rest))))))))
     (loop L)))

(define (find-list-element L pred success-cont failure-cont)
  (letrec ((loop (lambda (L)
                   (match L
                     ([] (failure-cont))
                     ((list-of x rest) (check ((pred x) (success-cont x))
                                              (else (loop rest))))))))
     (loop L)))

(define (list->string L sep)
  (match L
    ([] "")
    ([x] (val->string x))
    ((list-of x (bind rest (list-of _ _)))
      (join (val->string x)
            sep
            (list->string rest sep)))))

(define (weave x L)
  (letrec ((loop (lambda (front back results)
                   (match back
                     ([] (rev (add (join front [x]) results)))
                     ((list-of y rest) (loop (join front [y])
                                             rest
                                             (add (join front [x] back) results)))))))
     (loop [] L [])))


(define (cprod L1 L2)
  (let ((f (lambda (x)
             (map (lambda (y) [x y]) L2))))
   (flatten (map f L1))))

(define (all-list-elements L)
   (letrec ((loop (lambda (L results)
                    (match L
                      ([] (rev results))
                      ((list-of (some-list L') rest)
                          (let ((values (all-list-elements L')))
                             (loop rest (join (rev values) results))))
                      ((list-of x rest)
                         (loop rest (add x results)))))))
      (loop L [])))

(define (cprod* lists)
  (letrec ((loop (lambda (lists)
                   (match lists
                     ([L] L)
                     ((list-of L rest) (cprod L (loop rest)))))))
    (match lists
       ([L] (map (lambda (x) [x]) L))
       (_ (map all-list-elements (loop lists))))))
```

```
(define X cprod)

(define (cprod-k L1 L2 k)
  (check ((less? k 3) (cprod L1 L2))
         (else (let ((f (lambda (x)
                                  (map (lambda (y) (add x y))
                                       (cprod-k L1 L2 (minus k 1))))))
                 (flatten (map f L1))))))

(define (iterate f x k)
  (check ((less? k 1) x)
         (else (iterate f (f x) (minus k 1)))))


(define (cprods L)
  (letrec ((loop (lambda (lists)
                    (match lists
                       ([L] L)
                       ((list-of L more) (cprod L (loop more)))))))
    (let ((k (length L))
          (f (lambda (cp)
                 (map (lambda (L)
                         (flatten (map (lambda (x)
                                         (match x
                                           ((some-list _) x)
                                           (_ [x])))
                                       L)))
                      cp))))
      (match L
        ([only-list] [only-list])
        (_ (iterate f (loop L) k))))))

(define (all-distinct-pairs L)
  (let ((all-pairs (cprod L L)))
    (letrec ((loop (lambda (L res)
                     (match L
                       ([] res)
                       ((list-of (as p [x y]) more)
                          (check ((&& (negate (equal? x y)) (negate (member? [y x] res)))
                                   (loop more (add p res)))
                                 (else (loop more res))))))))
       (loop all-pairs [])))))


(define (occ-num x L)
  (letrec ((loop (lambda (L res)
                   (match L
                     ([] res)
                     (((list-of y more) where (equal? x y)) (loop more (plus res 1)))
                     ((list-of y more) (loop more res))))))
    (loop L 0)))


define (find-first-element M list) :=
  match list {
    (list-of first rest) =>
      try { (!M first) | (!find-first-element M rest) }
  }

define (list->counts L) :=
  letrec {loop := lambda (L M)
                    match L {
                      [] => M
                    | (list-of x rest) => (loop rest (Map.add M [[x try {(1 plus (M x)) | 1}]]))
                    }}
    (loop L (Map.make []))
```