

lib/basic/list.ath

```

1 (define (for-each L pred?)
2   (match L
3     ([] true)
4     ((list-of x xs) (&& (pred? x) (for-each xs pred?))))
5
6 (define (for-some L pred?)
7   (match L
8     ([] false)
9     ((list-of x xs) (|| (pred? x) (for-some xs pred?))))
10
11 (define (negate t)
12   (match t
13     (true false)
14     (false true)))
15
16 (define (for-none L pred?)
17   (for-each L (lambda (x) (negate (pred? x)))))
18
19 (define (member? x L)
20   (for-some L (lambda (y) (equal? x y))))
21
22 # The following returns the index of the first (leftmost) occurrence of x in L.
23 # If x does not occur in L, then false is returned. Index-counting starts with 1.
24
25 (define (member-index x L)
26   (letrec ((search (lambda (L i)
27                     (match L
28                       ([] false)
29                       ((list-of y rest) (check ((equal? x y) i)
30                                                (else (search rest (plus i 1)))))))
31     (search L 1)))
32
33 (define (member-eq? x L eq?)
34   (match L
35     ([] false)
36     ((list-of y rest) (check ((eq? x y) true)
37                              (else (member-eq? x rest eq?)))))
38
39 (define (subset? L1 L2)
40   (for-each L1 (lambda (x) (member? x L2))))
41
42 (define (subset-eq? L1 L2 eq)
43   (for-each L1 (lambda (x) (member-eq? x L2 eq))))
44
45 (define (equal-lists-as-sets L1 L2)
46   (&& (subset? L1 L2)
47        (subset? L2 L1)))
48
49 (define (for-some' list pred?)
50   (letrec ((loop (lambda (previous rest)
51                   (match rest
52                     ([] [])
53                     ((list-of x more) (check ((pred? x) [(rev previous) x more])
54                                                (else (loop (add x previous) more))))))
54     (loop [] list)))
55
56
57 (define (filter L f)
58   (letrec ((loop (lambda (L results)
59                   (match L
60                     ([] (rev results))
61                     ((list-of x rest) (check ((f x) (loop rest (add x results)))
62                                                (else (loop rest results))))))
63     (loop L [])))
64
65 (define (filter-out L f)
66   (filter L (lambda (x) (negate (f x)))))
67
68 (define (filter-and-complement L f)

```

```

69  (letrec ((loop (lambda (L sat unsat)
70                  (match L
71                    ([] [sat unsat])
72                    ((list-of x rest) (check ((f x) (loop rest (add x sat) unsat))
73                                              (else (loop rest sat (add x unsat))))))))
74    (loop L [] []))
75
76  (define (filter-and-complement-rev L f)
77    (letrec ((loop (lambda (L sat unsat)
78                    (match L
79                      ([] [(rev sat) (rev unsat)])
80                      ((list-of x rest) (check ((f x) (loop rest (add x sat) unsat))
81                                                (else (loop rest sat (add x unsat))))))))
82      (loop L [] []))
83
84  (define (map-proc f l)
85    (match l
86      ([] ())
87      ((list-of x rest) (seq (f x) (map-proc f rest)))))
88
89  (define iter map-proc)
90
91  (define (map-proc2 f l)
92    (letrec ((loop (lambda (L)
93                    (match L
94                      ([] ())
95                      ((list-of x rest) (seq (f x) (loop rest))))))
96      (loop l))
97
98  (define (list-diff L1 L2)
99    (let ((len1 (length L1))
100         (len2 (length L2)))
101      (check ((greater? (plus len1 len2) 100)
102              (let ((T (table len2))
103                    (_ (map-proc (lambda (x) (table-add T [x --> true])) L2))
104                    (in-L2 (lambda (x) (try (table-lookup T x) false)))
105                    (filter L1 (lambda (x) (negate (in-L2 x))))))
106              (else (filter L1 (lambda (x) (negate (member? x L2))))))))
107
108  (define (remove x L)
109    (letrec ((patch (lambda (front back)
110                     (match front
111                       ([] back)
112                       ((list-of y rest) (patch rest (add y back))))))
113      (loop (lambda (L front)
114              (match L
115                ((list-of (val-of x) rest) (patch front (loop rest [])))
116                ((list-of y rest) (loop rest (add y front)))
117                (_ front))))
118      (loop L []))
119
120
121  (define (remove x L)
122    (letrec ((patch (lambda (front back)
123                     (match front
124                       ([] back)
125                       ((list-of y rest) (patch rest (add y back))))))
126      (loop (lambda (L front)
127              (match L
128                ((list-of (val-of x) rest) (patch (rev front) (loop rest [])))
129                ((list-of y rest) (loop rest (add y front)))
130                (_ (rev front))))))
131      (loop L []))
132
133  (define (list-remove x L) (remove x L))
134
135  (define (remove-and-preserve-order a L)
136    (letrec ((loop (lambda (rest already-seen)
137                    (match rest
138                      ((list-of x more)

```

```

139         (check ((equal? x a) (loop more already-seen))
140               (else (loop more (add x already-seen)))))
141     (loop L []))
142   (loop L []))
143
144 (define list-remove remove-and-preserve-order)
145
146 (define (rd L)
147   (letrec ((loop (lambda (L res)
148                     (match L
149                       ((list-of x more) (check ((member? x res) (loop more res))
150                                                  (else (loop more (add x res)))))
151                       ([[] res]))))
152     (loop L []))
153
154 (define (rd-eq L equal?)
155   (letrec ((loop (lambda (L res)
156                     (match L
157                       ((list-of x more) (check ((member-eq? x res equal?) (loop more res))
158                                                  (else (loop more (add x res)))))
159                       ([[] res]))))
160     (loop L []))
161
162 (define (remove-duplicates L)
163   (let ((T (table 50))
164         (occurs-once (lambda (x)
165                        (try (table-lookup T x)
166                            (seq (table-add T [x false])
167                                true)))))
167     (filter L occurs-once)))
168
169 (define rd remove-duplicates)
170
171 (define remove-duplicates-eq rd-eq)
172
173 (define (map-with-index f L)
174   (letrec ((loop (lambda (L results i)
175                     (match L
176                       ([[] (rev results))
177                        ((list-of x rest) (loop rest (add (f x i) results) (plus i 1))))))
178     (loop L [] 1)))
179
180 (define (app-with-index f L)
181   (letrec ((loop (lambda (L i)
182                     (match L
183                       ([[] ()]
184                        ((list-of x rest) (seq (f x i) (loop rest (plus i 1)))))))
185     (loop L 1)))
186
187 (define (map2 f L1 L2)
188   (letrec ((loop (lambda (L1 L2 results)
189                     (match [L1 L2]
190                       ([[] _] (rev results))
191                       ([_ []] (rev results))
192                       ([ (list-of x xs) (list-of y ys) ]
                        (loop xs ys (add (f x y) results))))))
193     (loop L1 L2 []))
194
195 # (map2 add [1 2 3] [[4 5] [6 7] [8 9 10]]) = [[1 4 5] [2 6 7] [3 8 9 10]]
196 # (map2 add [1] [[4 5] [6 7] [8 9 10]]) = [[1 4 5]]
197 # (map2 add [] [[4 5] [6 7] [8 9 10]]) = []
198 # (map2 add [1 2 3] []) = []
199
200 # To test the above:
201 # (let ((a 'a) (b 'b) (c 'c) (d 'd) (e 'e) (f 'f))
202 #   (seq
203 #     (writeln-val (map2 add [1 2 3] [[4 5] [6 7] [8 9 10]]))
204 #     (writeln-val (map2 add [1] [[4 5] [6 7] [8 9 10]]))
205 #     (writeln-val (map2 add [] [[4 5] [6 7] [8 9 10]]))
206 #     (writeln-val (map2 add [1 2 3] [])))
207
208

```

```

209 define (non-empty? L) := (match L ((list-of _ _) true) ([] false))
210
211 (define (map-select f L pred)
212   (letrec ((loop (lambda (L results)
213     (match L
214       ([] (rev results))
215       ((list-of x rest) (let ((res (f x)))
216         (check ((pred res) (loop rest (add res results)))
217         (else (loop rest results)))))))
218     (loop L [])))
219
220 (define (map-select-2 f L pred)
221   (letrec ((loop (lambda (L results)
222     (match L
223       ([] (rev results))
224       ((list-of x rest) (check ((pred x) (loop rest (add (f x) results)))
225       (else (loop rest results))))))
226     (loop L [])))
227
228 (define map-list map)
229
230 (define (zip L1 L2)
231   (letrec ((f (lambda (L1 L2 res)
232     (match [L1 L2]
233       ([(list-of x1 rest1) (list-of x2 rest2)] (f rest1 rest2 (add [x1 x2] res)))
234       (_ (rev res)))))
235     (f L1 L2 [])))
236
237 (define list-zip zip)
238
239 (define (unzip L)
240   (letrec ((loop (lambda (L L1 L2)
241     (match L
242       ([] [(rev L1) (rev L2)])
243       ((list-of [x y] rest) (loop rest (add x L1) (add y L2)))))
244     (loop L [] [])))
245
246 ## Here the input method M is a regular method
247 ## that takes an input x (usually a sentence) and produces
248 ## a theorem. We apply M to all elements of the input list L,
249 ## collecting the results along the way, ultimately passing them
250 ## to the continuation K. Note that an error will result if
251 ## the application of M to an element of the list L fails.
252
253 (define (map-method M L K)
254   (dletrec ((loop (method (L res)
255     (dmatch L
256       ([] (!K (rev res)))
257       ((list-of x rest) (dlet ((th (!M x))
258         (!loop rest (add th res))))))
259     (!loop L [])))
260
261 ## Here the input method M is also a regular method
262 ## that takes an input x (usually a sentence) and produces
263 ## a theorem, but we don't fail if applying M to an
264 ## element of the list L produces an error. Conceivably,
265 ## M could fail on every element of L, in which case we
266 ## will simply pass the empty list to the continuation K:
267
268 (define (map-method-non-strictly M L K)
269   (dletrec ((loop (method (L res)
270     (dmatch L
271       ([] (!K (rev res)))
272       ((list-of x rest) (dlet ((ok? (cell false))
273         (th (dtry (dlet ((res (!M x))
274           (_ (set! ok? true)))
275           (!claim res))
276           (!true-intro)))
277         (res' (check ((ref ok?) (add th res))
278         (else res))))))

```

```

279                                     (!loop rest res'))))))))
280     (!loop L []))
281
282 (define (map-method-non-strictly-2 M L K)
283   (dletrec ((loop (method (L res productive)
284     (dmatch L
285       ([] (!K (rev res) productive))
286       ((list-of x rest) (dlet ((ok? (cell false))
287         (th (dtry (dlet ((res (!M x))
288           (_ (set! ok? true)))
289             (!claim res))
290             (!true-intro)))
291         ([res' productive'] (check ((ref ok?) [(add th res) (add x productive)]
292           (else [res productive]))))
293         (!loop rest res' productive'))))))))
294     (!loop L [] []))
295
296 ## map-methods-non-strictly takes a list of methods [M_1 ... M_k] and (non-strictly) applies
297 ## each M_i to the input list L, keeping track of the results along the way. All results
298 ## thereby obtained are ultimately passed to the continuation K:
299
300 (define (map-methods-non-strictly method-list L K)
301   (dletrec ((loop (method (methods all-results)
302     (dmatch methods
303       ([] (!K all-results))
304       ((list-of M rest) (!map-method-non-strictly M L
305         (method (new-results)
306           (!loop rest (join new-results all-results))))))
307     (!loop method-list [] []))
308
309
310
311 (define (map-methods-non-strictly-2 method-list L K)
312   (dletrec ((loop (method (methods all-results productive)
313     (dmatch methods
314       ([] (!K all-results (remove-duplicates productive)))
315       ((list-of M rest) (!map-method-non-strictly-2 M L
316         (method (new-results new-productive)
317           (!loop rest (join new-results all-results)
318             (join new-productive productive))))))
319     (!loop method-list [] []))
320
321 ## map-methods-non-strictly* is similar, but iterates this process up to a fixed
322 ## point, or until the number of iterations exceeds the limit specified by max-iterations,
323 ## whichever occurs first. (The parameter max-iterations is optional. The unit value,
324 ## or any other non-numeric value can be given to it, in which case the method will
325 ## iterate up to a fixed point.) Specifically, first we apply all methods to the input
326 ## list L, obtaining the first generation of results. Then we apply the methods to
327 ## those results, obtaining a second generation of results, and so on. A fixed point
328 ## is reached when the latest yield of results does not give us anything we haven't
329 ## already obtained in some previous generation. Finally, all the results are
330 ## cleaned up (removing duplicates, etc.) and passed to the continuation K:
331
332 (define (map-methods-non-strictly* method-list L K max-iterations)
333   (dlet ((test (match max-iterations
334     (x:Int (lambda (j) (greater? j max-iterations)))
335     _ (lambda (_) false))))
336     (dletrec ((loop (method (latest-results all-results i)
337       (!map-methods-non-strictly method-list latest-results
338         (method (new-results)
339           (dcheck ((| (test i) (subset? new-results all-results))
340             (!K (list-diff (remove-duplicates all-results) L)))
341           (else (!loop new-results (join new-results all-results) (plus i 1))))))
342       (!map-methods-non-strictly method-list L
343         (method (first-results)
344           (!loop first-results first-results 1))))
345
346 (define (map-methods-non-strictly-2* method-list L K max-iterations)
347   (dlet ((test (match max-iterations
348     (x:Int (lambda (j) (greater? j max-iterations)))

```

```

349      (_ (lambda (_) false))))))
350      (dletrec ((loop (method (latest-results all-results productive i)
351        (!map-methods-non-strictly-2 method-list latest-results
352          (method (new-results new-productive)
353            (dcheck ((|| (test i) (subset? new-results all-results))
354              (!K (list-diff (remove-duplicates all-results) (join L productive))))
355              (else (!loop new-results (join new-results all-results)
356                (join new-productive productive) (plus i 1))))))))))
357      (!map-methods-non-strictly-2 method-list L
358        (method (first-results productive)
359          (!loop first-results first-results productive 1))))))
360
361 ##
362 ## A "multi-method" is one that takes an input x and produces a *list*
363 ## of theorems. For that reason, a multi-method must be binary: it takes
364 ## a continuation K as a second argument, and passes the list of its results
365 ## to it when it is done. A typical example of a multi-method is a conjunction
366 ## elimination method that returns both the left and the right component
367 ## of a given conjunction:
368 ##
369 ##
370 ## (define (conj-elim p K)
371 ##   (dmatch p
372 ##     ((and _ _) (dlet ((p1 (!left-and p))
373 ##                       (p2 (!right-and p)))
374 ##       (!K [p1 p2])))
375 ##     (_ (!K []))))
376 ##
377 ## Note that multi-methods should be written to be fail-safe, so that
378 ## if an error occurs, the empty list of theorems is passed to the given
379 ## continuation (instead of halting execution with an error). This is
380 ## the reason why conj-elim was written as above with pattern matching,
381 ## rather than as follows:
382 ##
383 ## (define (conj-elim p K)
384 ##   (dlet ((p1 (!left-and p))
385 ##         (p2 (!right-and p)))
386 ##     (!K [p1 p2])))
387 ##
388 ## Of course, the first way of writing the method is not 100% fail-safe
389 ## either. E.g., if it's applied to a conjunction that is not in the a.b.,
390 ## we'll get an error. We can make a multi-method completely safe
391 ## as follows:
392 ##
393 ## (define (make-multi-method-safe M)
394 ##   (method (L K)
395 ##     (dtry (!M L K)
396 ##       (!K []))))
397 ##
398 ## However, use of make-multi-method-safe is inefficient and not recommended.
399 ##
400 ## "map-multi-method" below must be used to map a multi-method M to a list:
401 ##
402
403 (define (map-multi-method M L K)
404   (dletrec ((loop (method (L results)
405     (dmatch L
406       ([] (!K (rev results)))
407       ((list-of x rest) (!M x (method (new-results)
408         (!loop rest (join new-results results))))))))))
409     (!loop L [])))
410
411 ## A list method is one that can be applied to a list of inputs
412 ## and return a list of theorems. Thus a list method must be binary,
413 ## taking a continuation K as its second argument. The list of resulting
414 ## theorems is eventually passed to the continuation. Any regular method
415 ## (taking an input x and producing a theorem) can be turned into
416 ## a list method as follows:
417
418 (define (make-list-method M)

```

```

419 (method (L K)
420   (!map-method-non-strictly M L K)))
421
422
423 (define (map-multi-method* M L K max)
424   (dletrec ((loop (method (latest-results all-results i)
425     (!map-multi-method M latest-results
426       (method (new-results)
427         (dcheck (|| (subset? new-results all-results)
428           (less? max i)
429           (equal? max i))
430         (!K (remove-duplicates all-results)))
431       (else (!loop new-results (join new-results all-results) (plus i 1))))))))))
432   (!map-multi-method M L
433     (method (results)
434       (!loop results results 1))))))
435
436
437
438 (define (fold f lst id)
439   (match lst
440     ([] id)
441     ([x] (f x id))
442     ((list-of x (list-of y more))
443      (fold f (add (f x y) more) id))))
444
445 (define (foldl f e l)
446   (match l
447     ([] e)
448     ((list-of x xs) (foldl f (f e x) xs))))
449
450 (define (foldr f e l)
451   (match l
452     ([] e)
453     ((list-of x rest) (f x (foldr f e rest)))))
454
455 define (flatten L) := (foldl join [] L)
456
457
458 (define (nth i l)
459   (match [i l]
460     ([1 (list-of x _)] x)
461     (_ (nth (minus i 1) (tail l)))))
462
463 ## The above gets into an infinite loop for negative indices.
464 ## ith below fixes that. It also takes the list as the first
465 ## argument, the index
466
467 (define (ith L i)
468   (check ((greater? i 1) (ith (tail L) (minus i 1)))
469     ((equal? i 1) (head L))))
470
471 (define at ith)
472
473 (set-precedence at 107)
474
475 (define (nth i L) (ith L i))
476
477 (define (nth-tail l n)
478   (match n
479     (0 l)
480     (_ (nth-tail (tail l) (minus n 1)))))
481
482 (define (decompose-nth L n)
483   (letrec ((loop (lambda (L L' i ith)
484     (match [i L]
485       ([1 (list-of x rest)] (loop rest L' (minus i 1) x))
486       ([_ (list-of x rest)] (loop rest (add x L') (minus i 1) ith))
487       ([_ []] [ith L']))))))
488     (loop L [] n ())))

```

```

489
490 (define (prefix? s1 s2)
491   (match [s1 s2]
492     ([[] _] true)
493     ([ (list-of x rest1) (list-of x rest2)] (prefix? rest1 rest2))
494     (_ false)))
495
496 (define (find-min-rest l fun)
497   (letrec ((f (lambda (rem rest min i)
498     (match rem
499       ([[] [min rest]]
500         ((list-of x more) (check
501           ((greater? i 0) (check ((less? (fun x) (fun min)) (f more (add min rest)
502             x (plus i 1)))
503             (else (f more (add x rest) min (plus i 1))))))
504         (else (f more [] x (plus i 1)))))))
505     (check ((null? l) ()) (else (f l [] () 0)))))
506
507 (define (non-null L) (negate (null? L)))
508
509 (define (take l n)
510   (letrec ((f (lambda (l n res)
511     (match [l n]
512       ([_ 0] (rev res))
513       ([[] _] (rev res))
514       (_ (f (tail l) (minus n 1) (add (head l) res))))))
515     (f l n [])))
516
517 (define (drop L n)
518   (check ((less? n 1) L)
519     (else (drop (tail L) (minus n 1)))))
520
521 (define (split-list l n)
522   (letrec ((f (lambda (l n res)
523     (match [l n]
524       ([_ 0] [(rev res) l])
525       ([[] _] [(rev res) l])
526       (_ (f (tail l) (minus n 1) (add (head l) res))))))
527     (f l n [])))
528
529 (define (cut-in-half L)
530   (split-list L (div (length L) 2)))
531
532 (define (from-to i j)
533   (letrec ((loop (lambda (k res)
534     (check ((less? k i) res)
535       (else (loop (minus k 1) (add k res))))))
536     (check ((less? j i) [])
537       (else (loop j [])))))
538
539 (define to from-to)
540
541 (define (quant* Q var-list P)
542   (match var-list
543     ([[] P)
544     ((list-of v more-vars) (Q v (quant* Q more-vars P)))))
545
546 (define (forall* vl P) (quant* forall vl P))
547
548 (define (exists* vl P) (quant* exists vl P))
549
550 (define (close x)
551   (match x
552     ((some-sentence p) (forall* (free-vars p) p))
553     ((some-list L) (map close L))
554     (_ x)))
555
556 (define (make-selector-axioms S)
557   (let ((constructors (constructors-of S))
558     (selectors-of (lambda (con) (map-select-2 string->symbol

```



```

559 (selector-names con)
560 (lambda (str) (negate (null? str))))))
561
562 (make-straight-axioms
563   (lambda (con)
564     (let ((range (from-to 1 (arity-of con)))
565           (vars (map (lambda (_) (fresh-var)) range))
566           (con-term (make-term con vars))
567           (sel-vars (zip (selectors-of con) vars)))
568       (map (lambda (sel-var-pair)
569             (match sel-var-pair
570               ([sel v] (close (= (sel con-term) v))))
571             sel-vars))))
572   (make-option-axioms
573     (lambda (con)
574       (let ((range (from-to 1 (arity-of con)))
575             (vars (map (lambda (_) (fresh-var)) range))
576             (con-term (make-term con vars))
577             (selectors (selectors-of con))
578             (sel-vars (zip selectors vars))
579             (other-constructors (filter constructors (lambda (c) (negate (equal? c con)))))
580             (main-axioms (map (lambda (sel-var-pair)
581                               (match sel-var-pair
582                                 ([sel v] (close (= (sel con-term) (string->symbol "SOME") v))))
583                               sel-vars))
584             (option-axioms (lambda (sel)
585                               (map (lambda (c)
586                                     (let ((con-term (make-term c (map (lambda (_) (fresh-var)) (from-to 1 (arity-of c))
587                                                                           (close (= (sel con-term) (string->symbol "NONE")))))
588                                       other-constructors))))
589                               (all-option-axioms (flatten (map option-axioms selectors))))
590                               (join main-axioms all-option-axioms))))
591       (match (struc-with-option-valued-selectors? S)
592         (true (flatten (map make-option-axioms constructors)))
593         (_ (flatten (map make-straight-axioms constructors))))))
594
595 (define selector-axioms make-selector-axioms)
596
597 (define upper-case-alpha-char? c)
598   (&& (member? (compare-chars c 'A) ['greater 'equal])
599     (member? (compare-chars c 'Z) ['less 'equal]))
600
601 (define (downcase c)
602   (check ((upper-case-alpha-char? c) (char (plus (char-ord c) 32)))
603     (else c)))
604
605 (define (downcase-string str)
606   (map downcase str))
607
608 (define (separate L token)
609   (match L
610     ([ ] "")
611     ([s] s)
612     ((list-of s1 (bind rest (list-of _ _)))
613      (join s1 token (separate rest token)))))
614
615 (define (auto-induction-definition structure-name)
616   (let ((prequel (join "(define (" (downcase-string structure-name) "-induction goal-property)\n"))
617         (make-constructor-line
618           (lambda (c)
619             (let ((arity (arity-of c))
620                   (pattern (check ((less? arity 1)
621                                   (join "(bind state " (symbol->string c) ")")
622                                   (else (join "(bind state (" (symbol->string c) " " (separate (map (lambda (_) "_")
623                                                     (body (join "(!prove (goal-property state))\n"))
624                                                     (join "(" pattern " " body " "))))
625               (lines (separate (map make-constructor-line (constructors-of structure-name)) "\n"))
626               (cmd (join prequel "\n (by-induction (forall ?x (goal-property ?x))\n" lines " "))))
627             (process-input-from-string cmd)))
628

```

```

629
630
631
632 (make-private "auto-induction-definition")
633
634 # New additions, July 05 2015:
635
636 # (datatype (List-Of T)
637 #   Nil
638 #   (Cons hd:T tl:(List-Of T)))
639
640 #datatype (List T) := nil | (cons hd:T tl:(List T))
641
642 datatype (List T) := nil | (:: hd:T tl:(List T))
643
644 define [Nil Cons] := try { [nil ::] | [nil cons] }
645
646 (define (makeList L)
647   (match L
648     ([] Nil)
649     ((list-of x rest) (Cons x (makeList rest)))))
650
651
652 (define (alist->list L)
653   (match L
654     ([] (Nil))
655     ((list-of x rest) (Cons x (alist->list rest)))
656     (_ L)))
657
658
659 (define (alist->list inner)
660   (letrec ((loop (lambda (L)
661                     (match L
662                       ([] (Nil))
663                       ((list-of x rest) (Cons (inner x) (loop rest)))
664                       (_ L)))))
665     loop))
666
667 (define (list->alist L)
668   (match L
669     (Nil [])
670     ((Cons x rest) (add x (list->alist L)))
671     (_ L)))
672
673 (define (list->alist inner)
674   (letrec ((loop (lambda (L)
675                     (match L
676                       (Nil [])
677                       ((Cons x rest) (add (inner x) (loop rest)))
678                       (_ L)))))
679     loop))
680
681
682 (define
683   (even-positions L)
684   (match L
685     ((list-of _ rest) (odd-positions rest))
686     (_ []))
687   (odd-positions L)
688   (match L
689     ((list-of x rest) (add x (even-positions rest)))
690     (_ [])))
691
692 (define (merge L1 L2 less?)
693   (match [L1 L2]
694     ([ (list-of x rest1) (list-of y rest2) ]
695      (check ((less? x y) (add x (merge rest1 L2 less?)))
696             (else (add y (merge rest2 L1 less?))))))
697     ([[] _] L2)
698     (_ L1)))

```

```

699
700 (define (merge-sort' L less?)
701   (match L
702     ((list-of _ (list-of _ rest)) (merge (merge-sort' (odd-positions L) less?)
703                                           (merge-sort' (even-positions L) less?) less?))
704     (_ L)))
705
706 (define (merge-sort L less?)
707   (letrec ((sort (lambda (L)
708                     (match L
709                       ((list-of _ (list-of _ _))
710                        (let (([L1 L2] (cut-in-half L)))
711                          (merge (sort L1) (sort L2) less?)))
712                       (_ L))))))
713     (sort L)))
714
715 (define (skip-until list pred)
716   (match list
717     ([[] []])
718     ((list-of x rest) (check ((pred x) list)
719                               (else (skip-until rest pred))))))
720
721
722 (define (skip-until' list pred)
723   (letrec ((loop (lambda (L so-far)
724                     (match L
725                       ([[] [(rev so-far) []]])
726                       ((list-of c more)
727                        (check ((pred c) [(rev so-far) L])
728                              (else (loop more (add c so-far))))))))))
729     (loop list [])))
730
731
732 (define (tokenize L delims)
733   (let ((pred (lambda (c)
734                 (member? c delims)))
735         (pred' (lambda (c) (negate (member? c delims)))
736         (add' (lambda (str tokens)
737                 (check ((null? str) tokens)
738                       (else (add str tokens))))))
739     (letrec ((loop (lambda (L tokens)
740                       (match L
741                         ([[] (rev tokens)]
742                          (_ (let (([_ rest] (skip-until' L pred'))
743                              ([token more] (skip-until' rest pred)))
744                                (loop more (add' token tokens)))))))
745     (loop L [])))
746
747 (define tokenize-string tokenize)
748
749
750 # (first-image-that-works L f) returns (f x) for the first (leftmost) element x
751 # of L such that (f x) is defined (its computation doesn't lead to an error).
752 # If L has no such x, (first-image-that-works L f) fails in error:
753
754 define (first-image L f) :=
755   match L {
756     (list-of x rest) => try { (f x) | (first-image rest f) }
757   }
758
759 (define (find-element L pred success failure)
760   (letrec ((search (lambda (list)
761                       (match list
762                        ([[]] (failure))
763                        ((list-of x rest) (check ((pred x) (success x))
764                                                  (else (search rest)))))))
765     (search L)))
766
767 ## (find-element' L pred f success failure) finds the first element
768 ## x of L such that (pred (f x)) holds, if such x exists in L.

```

```

769 ## If so, then (success (f x)) is called, otherwise (failure) is called.
770
771 (define (find-element' L pred f success failure)
772   (letrec ((search (lambda (list)
773                     (match list
774                       ([] (failure))
775                       ((list-of x rest)
776                        (let ((res (f x)))
777                          (check ((pred res) (success res))
778                                (else (search rest))))))))
779     (search L)))
780
781 (define find-some-element find-element')
782
783 ## (find-first L f) finds the first (leftmost) member x of list L such that
784 ## (f x) returns something - call it y - other than false. That result, y,
785 ## is the result of (find-first L pred). If L contains no such x, then an error occurs:
786
787 (define (find-first L f)
788   (match L
789     ((list-of x rest) (match (f x)
790                             (false (find-first rest f))
791                             (res res))))
792
793 # find-first' works like find-first, except that if there is no element x in L
794 # such that (f x) is non-false, then the nullary continuation K is invoked:
795
796 (define (find-first' L f K)
797   (match L
798     ((list-of x rest) (match (f x)
799                             (false (find-first' rest f K))
800                             (res res)))
801     (_ (K))))
802
803
804
805 (define (find-list-member L pred? success failure)
806   (dmatch L
807     ([] (!failure))
808     ((list-of x rest) (dcheck ((pred? x) (!success x))
809                               (else (!find-list-member rest pred? success failure)))))
810
811
812 (define (find-some L M K)
813   (dmatch L
814     ([] (!K))
815     ((list-of x rest) (dtry (!M x)
816                             (!find-some rest M K))))
817
818
819 # (find-in-list L P) takes finds the first element y in L such
820 # that y has the unary property P. In addition, the lists to the
821 # left and right of y (in L) are also returned. E.g.,
822 # (find-in-list [0 6 3 4 10] odd?) returns [[0 6] 3 [4 10]].
823 # If no such y is found, the unit () is returned.
824
825 (define (find-in-list L P)
826   (letrec ((loop (lambda (L prefix)
827                   (match L
828                     ([] ())
829                     ((list-of x more) (check ((P x) [(rev prefix) x more])
830                                               (else (loop more (add x prefix)))))))
831     (loop L [])))
832
833 (define (find-decomposition L P)
834   (letrec ((loop (lambda (L prefix)
835                   (match L
836                     ([] ())
837                     ((list-of x more) (check ((P prefix x more) [prefix x more])
838                                               (else (loop more (join prefix [x]))))))))
838

```

```

839     (loop L []))
840
841
842 (define (max i j)
843   (check ((less? i j) j)
844     (else i)))
845
846 (define (min i j)
847   (check ((less? i j) i)
848     (else j)))
849
850 (define (min-or-max L seed f)
851   (match L
852     ([ seed)
853     ((list-of x rest) (min-or-max rest (f seed x) f))))
854
855 (define (generic-list-min L generic-min)
856   (match L
857     ([ ()]
858     (_ (min-or-max L (head L) generic-min))))
859
860 (define (list-max L)
861   (match L
862     ([ ()]
863     (_ (min-or-max L (head L) max))))
864
865 (define (list-min L)
866   (match L
867     ([ ()]
868     (_ (min-or-max L (head L) min))))
869
870
871 (define (max* L)
872   (match L
873     ([ 0]
874     (_ (list-max L))))
875
876 (define (intersection A B)
877   (letrec ((loop (lambda (A B result)
878     (match A
879       ([ result)
880       ((list-of x Amore)
881         (check ((&& (member? x B) (negate (member? x result)))
882           (loop Amore B (add x result)))
883         (else (loop Amore B result)))))))
884     (loop A B [])))
885
886
887 # define (fast-intersection A B) :=
888 #   let {T := (list->table A)}
889 #   (filter B
890 #     (lambda (x) (x HashTable.in T)))
891
892 # (define (fast-intersection* lists)
893 #   (match lists
894 #     ([ []]
895 #     ([L] L)
896 #     ((list-of L more)
897 #     (fast-intersection L (fast-intersection* more))))
898
899 (define list-intersection intersection)
900
901 (define (intersection* lists)
902   (match lists
903     ([ []]
904     ([L] L)
905     ((list-of L more)
906     (intersection L (intersection* more))))
907
908 (define (fast-intersection L1 L2)

```

```

909 (let ((T (table 10))
910       (_ (map-proc (lambda (x) (table-add T [x --> true])) L1)))
911   (filter L2 (lambda (x) (try (table-lookup T x) false)))))
912
913 (define (all-but-last L)
914   (match L
915     ([] [])
916     ([x] [])
917     ((list-of x rest) (add x (all-but-last rest)))))
918
919 (define (find L pred success-cont failure-cont)
920   (letrec ((loop (lambda (L)
921                     (match L
922                       ([] (failure-cont))
923                       ((list-of x rest) (check ((pred x) (success-cont x))
924                                                  (else (loop rest)))))))
925     (loop L)))
926
927 (define (find-list-element L pred success-cont failure-cont)
928   (letrec ((loop (lambda (L)
929                     (match L
930                       ([] (failure-cont))
931                       ((list-of x rest) (check ((pred x) (success-cont x))
932                                                  (else (loop rest)))))))
933     (loop L)))
934
935 (define (list->string L sep)
936   (match L
937     ([] "")
938     ([x] (val->string x))
939     ((list-of x (bind rest (list-of _ _)))
940      (join (val->string x)
941            sep
942            (list->string rest sep)))))
943
944 (define (weave x L)
945   (letrec ((loop (lambda (front back results)
946                     (match back
947                       ([] (rev (add (join front [x]) results)))
948                       ((list-of y rest) (loop (join front [y])
949                                                  rest
950                                                  (add (join front [x] back) results))))))
951     (loop [] L [])))
952
953
954 (define (cprod L1 L2)
955   (let ((f (lambda (x)
956              (map (lambda (y) [x y]) L2))))
957     (flatten (map f L1))))
958
959 (define (all-list-elements L)
960   (letrec ((loop (lambda (L results)
961                     (match L
962                       ([] (rev results))
963                       ((list-of (some-list L') rest)
964                        (let ((values (all-list-elements L')))
965                          (loop rest (join (rev values) results))))
966                       ((list-of x rest)
967                        (loop rest (add x results))))))
968     (loop L [])))
969
970 (define (cprod* lists)
971   (letrec ((loop (lambda (lists)
972                     (match lists
973                       ([L] L)
974                       ((list-of L rest) (cprod L (loop rest))))))
975     (match lists
976       ([L] (map (lambda (x) [x]) L))
977       (_ (map all-list-elements (loop lists)))))
978

```

```

979 (define X cprod)
980
981 (define (cprod-k L1 L2 k)
982   (check ((less? k 3) (cprod L1 L2))
983     (else (let ((f (lambda (x)
984       (map (lambda (y) (add x y))
985         (cprod-k L1 L2 (minus k 1))))))
986       (flatten (map f L1))))))
987
988 (define (iterate f x k)
989   (check ((less? k 1) x)
990     (else (iterate f (f x) (minus k 1)))))
991
992
993 (define (cprods L)
994   (letrec ((loop (lambda (lists)
995     (match lists
996       ([L] L)
997       ((list-of L more) (cprod L (loop more))))))
998     (let ((k (length L))
999       (f (lambda (cp)
1000         (map (lambda (L)
1001           (flatten (map (lambda (x)
1002             (match x
1003               ((some-list _) x)
1004               (_ [x]))
1005             L)))
1006           cp))))
1007       (match L
1008         ([only-list] [only-list])
1009         (_ (iterate f (loop L) k))))))
1010
1011 (define (all-distinct-pairs L)
1012   (let ((all-pairs (cprod L L)))
1013     (letrec ((loop (lambda (L res)
1014       (match L
1015         ([] res)
1016         ((list-of (as p [x y]) more)
1017           (check ((&& (negate (equal? x y)) (negate (member? [y x] res)))
1018             (loop more (add p res)))
1019           (else (loop more res))))))
1020       (loop all-pairs []))))
1021
1022
1023 (define (occ-num x L)
1024   (letrec ((loop (lambda (L res)
1025     (match L
1026       ([] res)
1027       ((list-of y more) where (equal? x y)) (loop more (plus res 1))
1028       ((list-of y more) (loop more res))))))
1029     (loop L 0)))
1030
1031
1032 (define (find-first-element M list) :=
1033   match list {
1034     (list-of first rest) =>
1035       try { (!M first) | (!find-first-element M rest) }
1036   }
1037
1038 (define (list->counts L) :=
1039   letrec {loop := lambda (L M)
1040     match L {
1041       [] => M
1042       | (list-of x rest) => (loop rest (Map.add M [[x try {(1 plus (M x)) | 1}]]))
1043     }}
1044     (loop L (Map.make []))

```