UNIVERSITÀ DEGLI STUDI DI FIRENZE
Facoltà di Scienze Matematiche, Fisiche e Naturali
Self Studies

Lab experiments

SOME LISP CODE WRITTEN STUDYING MASTERPIECE BOOKS

MASSIMO NOCENTINI

2011-2012

Massimo Nocentini: *Some Lisp code written studying masterpiece books,* Self Studies, © 2011-2012

CONTENTS

1

THE LITTLE SCHEMER

1.1 LISP FUNCTIONS' DEFINITIONS

```
(defun atomp (x)
  "This function return true if the argument X is an atom, nil
otherwise."
  (not (listp x)))

(defun latp (lst)
  "This function return true if the argument LST is a list that
doesn't contain any lists, nil otherwise."
  (cond
    ((null lst) t)
    ((atomp (car lst)) (latp (cdr lst)))
    (t nil) ) )

(defun memberp (a lat)
  "This function return true if the argument A is a member of the list
     LAT"
  (cond
    ((null lat) nil)
    (t (or (equal-sexps a
         (car lat))
     (memberp a (cdr lat))))))

(defun rember (a lat)
  (cond
    ((null lat) (quote ()))
    ((eq a (car lat)) (cdr lat))
    (t (cons      ;here we know that (car lat) is not
        ;equal to the element to remove, so we
        ;have to save it building a cons
        ;structure
       (car lat)
       (rember a (cdr lat))))))

(defun firsts (l)
  (cond ((null l) (quote ()))
  (t (cons (car (car l))     ;typical element
     (firsts (cdr l)))))) ;natural recursion

(defun insertR (new old lat)
  (cond ((null lat) (quote ()))
  ((eq old (car lat)) (cons old ;we could use (car lat) also,
          ;but this implies a
```

```
           ;computation, old is given
           ;and, in this case, is equals
           ;to (car lat)
           (cons new
           (cdr lat))))
  (t (cons (car lat)
     (insertR new old (cdr lat))))))

(defun insertL (new old lat)
  "Contract: atom atom list-of-atom -> list-of-atom

This function, given an element OLD to search, produce a new list such
that every occurrence of OLD is preceded by an occurrence of the atom
NEW."
  (cond ((null lat) (quote ()))
  ((eq old (car lat)) (cons new lat )) ;here we can use the
                ;entire lat as base for
                ;consing because (eq old
                ;(car lat)) is t
  (t (cons (car lat)
     (insertL new old (cdr lat))))))

(defun my-subst (new old lat)
    "Contract: atom atom list-of-atom -> list-of-atom

This function, given an atom OLD to research in a given list LAT,
return a new list such that the first occurrence of atom OLD is
substituted by the atom NEW."
  (cond ((null lat) (quote ()))
  ((eq old (car lat)) (cons new (cdr lat)))
  (t (cons (car lat)
     (my-subst new old (cdr lat))))))

(defun my-subst2 (new o1 o2 lat)
  (cond ((null lat) (quote ()))
  ((or (eq o1 (car lat))
       (eq o2 (car lat))) (cons new (cdr lat)))
  (t (cons (car lat)
     (my-subst2 new o1 o2 (cdr lat))))))

(defun multirember (a lat)
    "Contract: atom list-of-atom -> list-of-atom

This function, given an atom A and a list LAT, return a new list that
doesn't contain any atom A."
  (cond
    ((null lat) (quote ()))
    ((equal-sexps a (car lat))
     (multirember a (cdr lat))) ;we recur on the (cdr lat) because
        ;what we want is the remainder of the
        ;list such that doesn't contains any
        ;'a
    (t (cons      ;here we know that (car lat) is not
```

```
        ;equal to the element to remove, so we
        ;have to save it building a cons
        ;structure
       (car lat)
       (multirember a (cdr lat))))))

(defun multi-insert-r (new old lat)
  "Contract: atom atom list-of-atom -> list-of-atom

This function, given an atom OLD to research in a given list LAT,
return a new list that add the atom NEW to the right of every
occurrences of atom OLD."
  (cond ((null lat) (quote ()))   ;always the First Commandment
  ((eq old (car lat)) (cons old ;we could use (car lat) also,
          ;but this implies a
          ;computation, instead old is
          ;given and, in this branch, is
          ;equals to (car lat)
          (cons new
          (multi-insert-r new old ;inductive
                ;step
              (cdr lat)))))
  (t (cons (car lat)
     (multi-insert-r new old (cdr lat))))))

(defun multi-insert-l (new old lat)
  "Contract: atom atom list-of-atom -> list-of-atom

This function, given an atom OLD to research in a given list LAT,
return a new list that add the atom NEW to the left of every
occurrences of atom OLD."
  (cond ((null lat) (quote ()))
  ((eq old (car lat)) (cons new
          (cons old
          (multi-insert-l new
              old
              (cdr lat)))))
  (t (cons (car lat)
     (multi-insert-l new old (cdr lat))))))

(defun multi-subst (new old lat)
    "Contract: atom atom list-of-atom -> list-of-atom

This function, given an atom OLD to research in a given list LAT,
return a new list such that every occurrences of atom OLD is
substituted by the atom NEW."
  (cond ((null lat) (quote ()))
  ((eq old (car lat)) (cons new
          (multi-subst new
                 old
                 (cdr lat))))
  (t (cons (car lat)
     (multi-subst new old (cdr lat))))))
```

```
(defun o+ (n m)
  "Contract: number number -> number"
  (cond ((zerop m) n)
  (t (o+ (1+ n) (1- m)))) )

;; a subtle observation about this implementation and the one written
;; in the book: this version is tail-recursive, that is no work have
;; to be done when we reach the base of the recursion. In my head
;; there are two stack, one with n objects, the other with m
;; objects. Every recursive call pop an object from the second stack
;; and push it on the first one. When the second stack is empty the
;; recursion end and the result is already built in the first
;; stack. In the book version there is something more elengant in my
;; opinion: what is done is make a correspondence between the
;; recursive call with the knowledge of how many time we have to apply
;; the 1- function on the first argument. In other work we catch in
;; the computation (the recursive calls) a knowledge about data, so we
;; can forget the value of m during the work after the base of
;; recursion is reached. However that implementation is not tail
;; recursive.
(defun o- (n m)
  "Contract: number number -> number"
  (cond ((zerop m) n)
  (t (o- (1- n) (1- m)))) )

(defun addtup (tup)
  "Contract: tup-of-numbers -> number"
  (cond ((null tup) 0)
  (t (o+ (car tup) (addtup (cdr tup))))))

(defun x (n m)
  "Contract: number number -> number"
  (cond ((zerop m) 0)
  (t (o+ n (x n (1- m))))))

(defun tup+ (tup1 tup2)
  "Contract: list-of-numbers list-of-numbers -> list-of-numbers"
  (cond ((null tup1) tup2)
  ((null tup2) tup1)
  (t (cons (o+ (car tup1)
         (car tup2))
     (tup+ (cdr tup1)
         (cdr tup2))))))

(defun greater-than (n m)
  "Contract: number number -> boolean

Observation: this function require that the two parameters N and M are
non negative integers"
  (cond ((zerop n) nil)     ;if we reason only about whole
          ;integers than 0 is less than
          ;or equals to any other
```

```
            ;integer, so in all two cases
            ;the answer is false. Observe
            ;that if we swap the two
            ;questions, the function is
            ;not correct.
  ((zerop m) t)      ;if we ask this question, N
            ;must be some positive
            ;integer, hence it is always
            ;greater than M which, in this
            ;question is 0, so we answer
            ;true.
  (t (greater-than (1- n) (1- m))))) ;otherwise we natural recur
               ;on both numbers

(defun less-than (n m)
  "Contract: number number -> boolean
Observation: this function require that the two parameters N and M are
non negative integers"
  (cond ((zerop m) nil)           ;if m is zero than any other
            ;number n is at least equals
            ;or greater than, in both
            ;cases not less. Hence we
            ;answer false
  ((zerop n) t)      ;if we reason only about whole
            ;integers than 0 is less than
            ;or equals to any other
            ;integer, and if we ask this
            ;question then m is surely a
            ;positive integer, hence we
            ;answer t. Observe that if we
            ;swap the two questions, the
            ;function is not correct.
  (t (less-than (1- n) (1- m))))) ;otherwise we natural recur on
            ;both numbers

(defun our-equal (n m)
  (cond
    ((zerop m) (zerop n))
    ((zerop n) nil)      ;here we know that M isn't
            ;zero, so if N is zero surely
            ;they are different
    (t (our-equal (1- n) (1- m))) ;use natural recursion to find
            ;out the answer
    )
  )


(defun our-expt (base exponent)
  "Contract: number number -> number"
  (cond
    ((zerop exponent) 1)     ;by the fifth commandment we
            ;return 1 as termination value
            ;because we're building a
```

```
          ;number with 'x'
    (t (x base (expt base (1- exponent)))) ) )

(defun integer-division (dividend divisor)
  "Contract: number number -> number

How many times divisor is in dividend space?"
  (cond
    ((less-than dividend divisor) 0)    ;by the fifth
            ;commandment relative
            ;to 'addition'
    (t (1+ (integer-division (o- dividend divisor) divisor))) ))

(defun our-length (lat)
  "Contract: list-of-atom -> number"
  (cond
    ((null lat) 0)
    (t (1+ (our-length (cdr lat)))) ;here we know that at least an
          ;atom is present in the list
          ;LAT, so remember that with 1+
    ) )

(defun our-pick (n lat)
  "Contract: number list-of-atom -> atom"
  (cond
    ((zerop (1- n)) (car lat) )
    (t (our-pick (1- n) (cdr lat)))) )

(defun rempick (n lat)
  "Contract: number list-of-atom -> list-of-atom"
  (cond
    ((zerop (1- n)) (cdr lat))    ;we have to discard the car
          ;element because we've
          ;decremented N to the
          ;requested index
    (t (cons (car lat)
       (rempick (1- n) (cdr lat)))) ;use the natural recursion
            ;onto both N and LAT
    ) )

(defun no-nums (lat)
  "Contract: list-of-atom -> list-of-atom"
  (cond
    ((null lat) (quote ()))
    (t (cond
   ((numberp (car lat)) (no-nums (cdr lat)) )
   (t (cons (car lat) (no-nums (cdr lat)))))) ) )

(defun all-nums (lat)
  "Contract: list-of-atom -> tuple"
  (cond
    ((null lat) (quote ()))
    (t (cond
```

```lisp
    ((numberp (car lat)) (cons (car lat)
             (all-nums (cdr lat))))
    (t (all-nums (cdr lat)))))) )

(defun eqan (a1 a2)
  "Contract: atom atom -> boolean"
  (cond
    ((and (numberp a1) (numberp a2)) (our-equal a1 a2))
    ((or (numberp a1) (numberp a2)) nil) ;if at least one of them is a
           ;number, so they are not the
           ;same atom
    (t (eq a1 a2))) )

(defun occur (a lat)
  "Contract: atom list-of-atom -> number"
  (cond
    ((null lat) 0)      ;by first commandment follow
          ;the condition on the argument
          ;that change during
          ;recursion. By the Fifth
          ;commandment we return 0
          ;because we're building a
          ;number with the operator +
    (t (cond
   ((eqan a (car lat)) (1+ (occur a (cdr lat))))
   (t (occur a (cdr lat))))) ) )

(defun onep (n)
  "Contract: number -> boolean"
  (zerop (1- n)) )

(defun rempick-using-one (n lat)
  "Contract: number list-of-atom -> list-of-atom"
  (cond
    ((onep n) (cdr lat))
    (t (cons (car lat) (rempick-using-one (1- n) (cdr lat))))))

(defun rember* (a l)
  "Contract: atom list-of-sexp -> list-of-sexp"
  (cond
    ((null l) (quote ()))   ;if the list is empty we have
          ;nothing to remove, and by
          ;Fifth Commandment we return
          ;() because we are building a
          ;list with the operator cons.
    ((atomp (car l))      ;if the car of L is an atom we
          ;are able to perform a check
          ;with the given A to remove
     (cond
       ((eqan a (car l)) (rember* a (cdr l))) ;if the car is really A
             ;we return what return
             ;this function applied
             ;to the cdr of L
```

```
                 ;(assuming by induction
                 ;that this function is
                 ;correct for cdr of
                 ;lists, this step is
                 ;what really remove the
                 ;atom a
        (t (cons (car l) (rember* a (cdr l)))))) ;otherwise we keep the
            ;atom A, consing it on
            ;the list without atom
            ;A by induction hp
     (t (cons (rember* a (car l))     ;if the car of L isn't
            ;an atom it must be a
            ;list, so we rebuilt
            ;the cons structure
            ;(because a list is at
            ;the end a cons
            ;structure) with the
            ;two recursive
            ;applications of these
            ;rules (another way to
            ;call a (this)
            ;function)
        (rember* a (cdr l)))))
  )

(defun insert-r* (new old l)
  "Contract: atom atom list-of-sexp -> list-of-sexp"
  (cond
     ((null l) (quote ()))    ;by the first commandment ask
          ;for null when we recur
          ;modifying a list
          ;parameter. By the fifth
          ;commandment return () because
          ;we have to build lists with
          ;cons
     ((atomp (car l))        ;we ask if the car is an atom,
          ;in this case...
      (cond
        ((eqan old (car l)) (cons old
          (cons new
                (insert-r* new old (cdr l)))))
        (t (cons (car l)
     (insert-r* new old (cdr l))))))
     (t (cons (insert-r* new old (car l))
        (insert-r* new old (cdr l)))) ) )

(defun occur* (a l)
  "Contract: atom list-of-sexp -> number"
  (cond
     ((null l) 0)       ;by the First Commandment we
          ;check for termination. By the
          ;Fifth Commandment we return 0
          ;because we have to build a
```

```
          ;number with the operator +
    ((atomp (car l))
     (cond
       ((eqan a (car l)) (1+ (occur* a (cdr l))))
       (t (occur* a (cdr l)))))
    (t (o+ (occur* a (car l))
     (occur* a (cdr l)))) ))

(defun subst* (new old l)
  "Contract: atom atom list-of-sexp -> list-of-sexp"
  (cond
    ((null l) (quote ()))   ;by first and fifth
          ;commandment
    ((atomp (car l))
     (cond
       ((eqan old (car l)) (cons new
         (subst* new old (cdr l))))
       (t (cons (car l)
    (subst* new old (cdr l))))))
    (t (cons (subst* new old (car l))
       (subst* new old (cdr l))))
    )
  )

(defun insert-l* (new old l)
  "Contract: atom atom list-of-sexp -> list-of-sexp"
  (cond
    ((null l) (quote ()))   ;guard by first commandment,
          ;return by the fifth
          ;commandment
    ((atomp (car l))
     (cond
       ((eqan old (car l)) (cons new
         (cons old ;here we have to cons
             ;because we have to change
             ;at least one argument when
             ;we recur: in our case this
             ;argument is L
               (insert-l* new old (cdr l)))))
       (t (cons (car l)
    (insert-l* new old (cdr l))))))
    (t (cons (insert-l* new old (car l))
       (insert-l* new old (cdr l))))
    ))

(defun member* (a l)
  "Contract: atom list-of-sexp -> boolean"
  (cond
    ((null l) nil)
    ((atomp (car l)) (or (eqan a (car l))
       (member* a (cdr l))) )
    (t (or (member* a (car l))
     (member* a (cdr l))))) )
```

```
(defun leftmost (l)
  "Contract: list-of-sexp -> atom"
  (cond
    ((atom (car l)) (car l))
    (t (leftmost (car l)))) )

(defun eqlistp-first-revision (l1 l2)
  "Contract: list-of-sexp list-of-sexp -> boolean"
  (cond
    ((and (null l1) (null l2)) t)
    ((and (null l1) (atomp l2)) nil)
    ((null l1) nil)       ;the complete form of the
          ;question should have been:
          ;(and (null l1) (listp l2))

    ;; if we reach the following question then l1 must be a non-empty
    ;; list, but we don't know about the nature of l2
    ((and (atomp (car l1)) (null l2)) nil)

    ;; if we reach the following question then we know: if the car of
    ;; L1 is an atom then l2 must be a list with at least one element
    ;; (otherwise the above question should have been
    ;; answered). Otherwise, if the car of L1 is a list, we don't know
    ;; the nature of l2
    ((and (atomp (car l1)) (atomp (car l2)))
     (and (eqan (car l1) (car l2))
    (eqlistp (cdr l1) (cdr l2))))
    ((atomp (car l1)) nil)    ; if the car of L1 is an atom
          ; then l2 must be a list with
          ; its car is also a list, so
          ; l1 is different from l2
    ;; at this point we know that the car of L1 is a list, hence now
    ;; we ask on the nature of l2
    ((null l2) nil)
    ((atomp (car l2)) nil)
    (t (and (eqlistp (car l1) (car l2))
      (eqlistp (cdr l1) (cdr l2))))
    ) )

(defun eqlistp-second-revision (l1 l2)
  "Contract: list-of-sexp list-of-sexp -> boolean"
  (cond
    ((and (null l1) (null l2)) t)
    ((or (null l1) (null l2)) nil)

    ;; if we reach the following question then we know: if the car of
    ;; L1 is an atom then l2 must be a list with at least one element
    ;; (otherwise the above question should have been
    ;; answered). Otherwise, if the car of L1 is a list, we don't know
    ;; the nature of l2
    ((and (atomp (car l1)) (atomp (car l2)))
     (and (eqan (car l1) (car l2))
```

```
    (eqlistp (cdr l1) (cdr l2))))

    ((or (atomp (car l1)) (atomp (car l2))) nil)

    (t (and (eqlistp (car l1) (car l2))
      (eqlistp (cdr l1) (cdr l2))))
    ) )

(defun equal-sexps (s1 s2)
  "Contract: sexp sexp -> boolean"
  (cond
    ((and (atomp s1) (atomp s2)) (eqan s1 s2))
    ((or (atomp s1) (atomp s2)) nil)
    (t (eqlistp s1 s2)) ) )

(defun eqlistp (l1 l2)
  "Contract: list-of-sexp list-of-sexp -> boolean"
  (cond
    ((and (null l1) (null l2)) t)
    ((or (null l1) (null l2)) nil)
    (t (and (equal-sexps (car l1) (car l2))
      (eqlistp (cdr l1) (cdr l2)))) ) )


(defun rember-sexp-version (sexp l)
  "Contract: sexp list-of-sexp -> list-of-sexp"
  (cond
    ((null l) (quote ()))
    ((equal-sexps sexp (car l)) (cdr l))
    (t (cons (car l) (rember-sexp-version sexp (cdr l))))))

(defun numberedp (aexp)
  "Contract: arithmetic-expression -> boolean"
  (cond
    ((atomp aexp) (numberp aexp))
    (t (and (numberedp (car aexp)) (numberedp (car
                    (cdr
              (cdr aexp)))))) ) )

(defun 1st-sub-exp (aexp)
  (car (cdr aexp)) )

(defun 2nd-sub-exp (aexp)
  (car (cdr (cdr aexp))) )

(defun operator (aexp)
  (car aexp) )

;; using help function to hide representation we are able to focus on
;; the recursive definition of our concept. The help function allow in
;; a second moment to use the representation that is the more suitable
;; for the needs. We can therefore write a triple of help functions
;; for each representation that we have to deal with.
```

```
(defun tls-value (nexp)
  "Contract: numbered-expression -> number"
  (cond
    ((atomp nexp) nexp)      ;we can return the nexp
          ;because it is an atom and, by
          ;contract, it is also a number
          ;so it is its value too.
    (t (funcall (atom-to-function (operator nexp))
    (tls-value (1st-sub-exp nexp))
    (tls-value (2nd-sub-exp nexp)))) ) )

(defun serop (n)
  "Contract: list-of-empty-lists -> boolean"
  (null n))

(defun edd1 (n)
  "contract: list-of-empty-lists -> list-of-empty-lists"
  (cons '() n))

(defun zub1 (n)
  "contract: list-of-empty-lists -> list-of-empty-lists"
  (cdr n) )

(defun os+ (n m)
  "contract: list-of-empty-lists list-of-empty-lists ->
      list-of-empty-lists"
  (cond
    ((serop m) n)
    (t (edd1 (os+ n (zub1 m)))) ) )

(defun setp (lat)
  "contract: list-of-atom -> boolean"
  (cond
    ((null lat) t)
    ((memberp (car lat) (cdr lat)) nil)
    (t (setp (cdr lat)))))

(defun makeset (lat)
  "contract: list-of-atom -> list-of-atom

This variation use the function MEMBERP as help function."
  (cond
    ((null lat) (quote ()))
    ((memberp (car lat) (cdr lat)) (makeset (cdr lat)) )
    (t (cons (car lat) (makeset (cdr lat)))) ) )

(defun makeset-variation (lat)
  "contract: list-of-atom -> list-of-atom

This variation use the function MULTIREMBER as help function."
  (cond
    ((null lat) (quote ()))
    (t (cons (car lat) (makeset-variation
```

```
      (multirember (car lat)
             lat) ;here we can improve using
            ;(cdr lat) to make one less
            ;remove, that is the (car
            ;lat) itself
      ))) ) )

(defun tls-subsetp (set1 set2)
  "contract: set set -> boolean"
  (cond
    ((null set1) t)     ;because the empty set is a
          ;subset of every set
          ;(included, of course, the
          ;empty set itself!)
    (t (and (memberp (car set1) set2)
      (tls-subsetp (cdr set1) set2))) ) )

(defun eqsetp (set1 set2)
  "contract: set set -> boolean"
  (and (tls-subsetp set1 set2)
       (tls-subsetp set2 set1))
  )

(defun intersectp (set1 set2)
  "contract: set set -> boolean"
  (cond
    ((null set1) nil)     ;an empty set hasn't any
          ;element that can share with
          ;any other set
    (t (or (memberp (car set1) set2)
     (intersectp (cdr set1) set2))) ) )

(defun intersect (set1 set2)
  "contract: set set -> set"
  (cond
    ((null set1) (quote ()))
    ((memberp (car set1) set2) (cons (car set1)
             (intersect (cdr set1) set2)))
    (t (intersect (cdr set1) set2))) )

(defun tls-union (set1 set2)
  "contract: set set -> set"
  (cond
    ((null set1) set2)
    ((memberp (car set1) set2) (tls-union (cdr set1) set2))
    (t (cons (car set1) (tls-union (cdr set1) set2))) ) )

(defun tls-set-difference (set1 set2)
  "contract: set set -> set"
  (cond
    ((null set1) (quote ()))
    ((memberp (car set1) set2) (tls-set-difference (cdr set1) set2))
    (t (cons (car set1) (tls-set-difference (cdr set1) set2))) ) )
```

```
(defun intersect-all (l-set)
  "contract: list-of-set -> list-of-atom"
  (cond
    ((null (cdr l-set)) (car l-set))  ;(car l-set) is a set by contract
    (t (intersect (car l-set) (intersect-all (cdr l-set)))) ) )

(defun pairp (sexp)
  "contract: list-of-sexp -> boolean"
  (cond
    ((atomp sexp) nil)      ;this condition is necessary
          ;because the recursive
          ;definition of sexp allow to
          ;be an atom or a list of sexp
    ((null sexp) nil)     ;if this is true then sexp is
          ;the empty list
    ((null (cdr sexp)) nil)   ;if this is true then sexp
          ;contains only one element
    ((null (cdr (cdr sexp))) t)   ;if this is true then sexp is
          ;really a pair
    (t nil) ))        ;otherwise sexp have at least
          ;three elements

(defun pair-first-component (pair)
  "contract: pair -> sexp"
  (car pair) )

(defun pair-second-component (pair)
  "contract: pair -> sexp"
  (car (cdr pair)) )

(defun pair-third-component (pair)
  "contract: pair -> sexp"
  (car (cdr (cdr pair))) )

(defun build-pair (s1 s2)
  "contract: sexp sexp -> pair"
  (cons s1 (cons s2 (quote ()))))

(defun relationp (l-sexp)
  "contract: list-of-sexp -> boolean"
  (and (setp l-sexp) (all-pair-in-list-p l-sexp))
  )

(defun all-pair-in-list-p (l-sexp)
  "contract: list-of-sexp -> boolean"
  (cond
    ((null l-sexp) t)
    (t (and (pairp (car l-sexp)) (all-pair-in-list-p (cdr l-sexp)))) )
    )

(defun tls-functionp (rel)
  "contract: relation -> boolean"
```

```
  (setp (firsts rel)) )

(defun revrel (rel)
  "contract: relation -> relation"
  (cond
    ((null rel) (quote ()))
    (t (cons (revpair (car rel))
       (revrel (cdr rel)))) ) )

(defun revpair (pair)
  "contract: pair -> pair"
  (build-pair (pair-second-component pair)
        (pair-first-component pair)) )

(defun fullfunp (fun)
  "contract: function -> boolean"
  (tls-functionp (revrel fun)) )

(defun rember-f (test-function)
  "contract: (lambda: sexp sexp -> boolean) ->
                 (lambda: sexp list-of-sexp -> list-of-sexp)"
  (function
   (lambda (a l)
    (cond
      ((null l) (quote ()))
      ((funcall test-function a (car l)) (cdr l))
      (t (cons (car l)
         (funcall (rember-f test-function) a (cdr l))))))))

(defun eq?-c (a)
  "contract: atom -> (lambda: atom -> boolean)"
  (function (lambda (x) (eq x a))) )

(defun insert-l-f (test-function)
  "contract: (lambda: sexp sexp -> boolean) -> (lambda: sexp sexp
list-of-sexp -> list-of-sexp)"
  (insert-g test-function
      (lambda (new old l)
  "contract: sexp sexp list-of-sexp -> list-of-sexp"
  (cons new (cons old l)) )))


;; this functions that takes a lambda (aka: a function) as parameter,
;; return some lambdas and NOT take a mix of atoms, lists and
;; lambdas. They take lambdas and return lambdas, no mix with other
;; elements.
(defun insert-r-f (test-function)
  "contract: (lambda: sexp sexp -> boolean) -> (lambda: sexp sexp
list-of-sexp -> list-of-sexp)"
  (insert-g test-function
      (lambda (new old l)
  "contract: sexp sexp list-of-sexp -> list-of-sexp"
  (cons old (cons new l)) )))
```

```
(defun insert-g (test-lambda consing-lambda)
  "contract: (lambda: sexp sexp -> boolean) (lambda: sexp sexp
list-of-sexp -> list-of-sexp) -> (lambda: sexp sexp list-of-sexp ->
list-of-sexp)"
  (function (lambda (new old l)
    (cond
      ((null l) (quote ()))
      ((funcall test-lambda old (car l))
       (funcall consing-lambda new old (cdr l)))
      (t (cons (car l)
         (funcall (insert-g test-lambda consing-lambda) new
      old (cdr l)))) ) ) ))

(defun subst-f (test-lambda)
  "contract: (lambda: sexp sexp -> boolean) -> (lambda: sexp sexp
  list-of-sexp -> list-of-sexp)"
  (insert-g test-lambda (lambda (new old l)
        (cons new l))) )

(defun rember-f-final (test-lambda)
  "contract: (lambda: sexp sexp -> boolean) -> (lambda: sexp sexp
  list-of-sexp -> list-of-sexp)"
  (insert-g test-lambda (lambda (new old l) l) ;just return the list,
                  ;ignore NEW and OLD
      ) )

(defun atom-to-function (atom)
  "Contract: atom -> (lambda: number number -> number)"
  (cond
    ((eq atom (quote +)) (function o+))
    ((eq atom (quote x)) (function x))
    (t (function expt))) )

(defun multi-rember-f (test)
  "contract: (lambda: sexp sexp -> boolean) -> (lambda: sexp
  list-of-sexp -> list-of-sexp)"
  (lambda (a lat)
    (cond
      ((null lat) (quote ()))
      ((funcall test a (car lat)) (funcall (multi-rember-f test)
             a (cdr lat)))
      (t (cons (car lat)
         (funcall (multi-rember-f test) a (cdr lat)))) )) )

(defun multi-rember-t (test l)
  "contract: (lambda: sexp -> boolean) list-of-sexp -> list-of-sexp"
  (cond
    ((null l) (quote ()))
    ((funcall test (car l)) (multi-rember-t test (cdr l)))
    (t (cons (car l) (multi-rember-t test (cdr l))))) )

(defun multi-rember-single-param (test)
```

```lisp
  "contract: (lambda: sexp -> boolean) -> (lambda: list-of-sexp ->
  list-of-sexp)"
  (lambda (l)
    (cond
      ((null l) (quote ()))
      ((funcall test (car l))   ;here we do Curry-ing on the
          ;function TEST: this function
          ;is passed as argument in
          ;multi-rember-single-param
          ;function, it isn't defined in
          ;the current lambda definition
       (funcall (multi-rember-single-param test)
    (cdr l)))
      (t (cons (car l)
         (funcall (multi-rember-single-param test)
      (cdr l)))))) )

(defun multi-rember&co (a lat col)
  "contract: atom list-of-atom (lambda: list-of-atom list-of-atom ->
  object) -> object"
  (cond
    ((null lat) (funcall col
       (quote ())
       (quote ())))
    ((eq a (car lat)) (multi-rember&co a
                (cdr lat)
                (lambda (newl seen)
           (funcall col
              newl
              (cons a seen)))))
    (t (multi-rember&co a
      (cdr lat)
      (lambda (newl seen)
        (funcall col
           (cons (car lat) newl)
           seen)))) ) )

(defun multi-insert-lr (new oldl oldr lat)
  "contract: atom atom atom list-of-atom -> list-of-atom"
  (cond
    ((null lat) (quote ()))
    ((eq (car lat) oldl) (cons new
             (cons oldl
             (multi-insert-lr new oldl
                  oldr (cdr
                  lat)))))
    ((eq (car lat) oldr) (cons oldr
             (cons new
             (multi-insert-lr new oldl
                  oldr (cdr
                  lat)))))
    (t (cons (car lat) (multi-insert-lr new oldl oldr (cdr lat)))) ) )
```

```
(defun multi-insert-lr&co (new oldl oldr lat col) ;label:**
  "contract: atom atom atom list-of-atom (lambda: list-of-atom number
number object) -> object"
  (cond
    ((null lat) (funcall col (quote ()) 0 0))
    ((eq (car lat) oldl)
     (multi-insert-lr&co    ;label:*
      new oldl oldr (cdr lat)
      (lambda (newlat l r)    ;these argument are the
          ;results of the induction
          ;call, that is the recursion
          ;invocation of ref:*
  (funcall col      ;here we invoke the collector
          ;for the actual computation
          ;ref:**, consing and adding
          ;something because in this
          ;cond's branch we know
          ;something (in this case that
          ;the (eq (car lat) oldl)),
          ;hence we cons the appropriate
          ;sequence of oldl and new and
          ;increment the left-insertion
          ;counter
     (cons new (cons oldl newlat))
     (1+ l)
     r))))
    ((eq (car lat) oldr)
     (multi-insert-lr&co
      new oldl oldr (cdr lat)
      (lambda (newlat l r)
  (funcall col
     (cons oldr (cons new newlat))
     l
     (1+ r)))) )
    (t (multi-insert-lr&co
  new oldl oldr (cdr lat)
  (lambda (newlat l r)
    (funcall col
      (cons (car lat) newlat) l r)))) ) )

(defun tls-evenp (n)
  "contract: number -> boolean"
  (our-equal (x (integer-division n 2) 2) n) )

(defun evens-only* (l)
  "contract: list-of-sexp -> list-of-sexp"
  (cond
    ((null l) (quote ()))
    ((atomp (car l))
     (cond
       ((tls-evenp (car l)) (cons (car l) (evens-only* (cdr l))))
       (t (evens-only* (cdr l)))))
    (t (cons (evens-only* (car l))
```

```
      (evens-only* (cdr l)))) ) )

(defun evens-only*&co (l col)
  "contract: list-of-sexp (lambda: list-of-sexp number number ->
  object) -> object"
  (cond
    ((null l) (funcall col (quote ()) 1 0))
    ((atomp (car l))
     (cond
       ((tls-evenp (car l))
  (evens-only*&co (cdr l)
      (lambda (newl prod sum)
        (funcall col
           (cons (car l) newl)
           (x prod (car l))
          sum))))
       (t (evens-only*&co (cdr l)
        (lambda (newl prod sum)
          (funcall col
             newl
             prod
             (o+ sum (car l))))))))
    (t (evens-only*&co (car l)
           (lambda (car-list car-prod car-sum)
       (evens-only*&co (cdr l)
           (lambda (cdr-list cdr-prod cdr-sum)
             (funcall col
                (cons car-list cdr-list)
                (x car-prod cdr-prod)
                (o+ car-sum
              cdr-sum))))))))
              )

(defun looking (a lat)
  "contract: atom list-of-atom -> boolean"
  (start-looking a (make-getting lat)))

(defun start-looking (a getting-lambda)
  "contract: atom (lambda: number -> atom) -> boolean"
  (keep-looking a (funcall getting-lambda 1) getting-lambda) )

(defun make-getting (lat)
  "contract: list-of-atom -> (lambda: number -> atom)

The lambda returned do curry-ing on the LAT argument. In this way we
hide the collection we take elements from"
  (lambda (number)
    (our-pick number lat)))

;; from this definition we see that lat is used only in one branch of
;; cond questions. We can refactor it, passing a function that hide
;; the argument LAT to the rest of the function (we'll remove it from
;; the argument list)
```

```
(defun keep-looking-to-refactor (a sorn lat)
  (cond
    ((numberp sorn) (keep-looking a (our-pick sorn lat) lat))
    (t (eq a sorn))))

(defun keep-looking (a sorn getting)
  "contract: atom atom (lambda: number -> atom) -> boolean"
  (cond
    ((numberp sorn) (keep-looking a (funcall getting sorn) getting))
    (t (eq a sorn)) ) )

(defun eternity (x)
  (eternity x))

(defun shift-pair (p)
  "contract: pair -> pair
such that (pair-first-component p) is a pair"
  (build-pair (pair-first-component (pair-first-component p))
        (build-pair (pair-second-component (pair-first-component p))
        (pair-second-component p))) )

(defun align (pora)
  "contract: (pair | atom) -> pair"
  (cond
    ((atomp pora) pora)
    ((pairp (pair-first-component pora)) (align (shift-pair pora)))
    (t (build-pair (pair-first-component pora)
       (align (pair-second-component pora)))) ) )

(defun count-atoms-in-pair (pair)
  "contract: pair -> number"
  (cond
    ((atomp pair) 1)
    (t (o+ (count-atoms-in-pair (pair-first-component pair))
     (count-atoms-in-pair (pair-second-component pair))))) )

(defun pair-weight (pair)
  "contract: pair -> number"
  (cond
    ((atom pair) 1)
    (t (o+ (x 2 (pair-weight (pair-first-component pair)))
     (pair-weight (pair-second-component pair))))) )

(defun pair-shuffle (pora)
  "contract: (pair | atom) -> pair"
  (cond
    ((atomp pora) pora)
    ((pairp (pair-first-component pora)) (pair-shuffle (revpair pora)))
    (t (build-pair (pair-first-component pora)
       (pair-shuffle (pair-second-component pora)))) ) )

(defun collatz (n col)
  "contract: number (lambda: number list-of-number -> object) ->
```

```
object"
  (cond
    ((onep n) (funcall col 1 (cons n (quote ())))) ;in this case we
                   ;have only the
                   ;current invocation
    ((tls-evenp n) (collatz (integer-division n 2)
          (lambda (times seen-numbers)
            (funcall col (1+ times) (cons n seen-numbers)))))
    (t (collatz (1+ (x n 3))
    (lambda (times seen-numbers)
      (funcall col (1+ times) (cons n seen-numbers)))))) )

;; ... a(1 59) c(1 58) (0 59) (0 . 60) ...
;; ... a(1 59) b(0 60) (0 . 61) ...

;; in order to compute the Ackermann value for the pair a(1 59) we
;; need to look at its right first (mimic the bottom definition) which
;; in turn needs the value of c(1 58). Hence for a(1 59) we need b(0
;; 60) which, in nested fashion, needs c(1 58).
(defun ackermann (n m col)
  "contract: number number (lambda: number list-of-pair -> object) ->
object"
  (cond
    ((zerop n) (progn (funcall col 1 (cons (build-pair n m)
                 (cons (cons n (1+ m))
                 (quote ()))))
          (1+ m)) )
    ((zerop m) (ackermann (1- n) 1
        (lambda (times pairs)
          (funcall col
             (1+ times)
             (cons (build-pair n m)
             pairs)))))
    (t (ackermann (1- n) (ackermann n (1- m)
            (lambda (times pairs)
              (funcall col
                 (1+ times)
                 (cons (build-pair n m)
                 pairs))))
      (lambda (times pairs)
        (funcall col
           (1+ times)
           (cons (build-pair n m) pairs))))) ) )

(defun eternity-length (l)
  "contract: list-of-sexp -> number"
  (cond
    ((null l) 0)
    (t (1+ (funcall
      (lambda (l)       ;this lambda computes
            ;the length of a
            ;function with at most
            ;one element
```

```
        (cond
    ((null l) 0)
    (t (1+ (funcall
      (lambda (l)     ;here we replace the
           ;definition of
           ;length-0 because it
           ;cannot be defuned
        (cond
          ((null l) 0)
          (t (1+ (eternity
            (cdr l))))))
      (cdr l)) ))))
      (cdr l)) ))) )

(defun length-lambda-factory (length-lambda)
  "contract: (lambda: list-of-sexp -> number) -> (lambda: list-of-sexp
  -> number)"
  (lambda (l)
    (cond
      ((null l) 0)
      (t (1+ (funcall length-lambda (cdr l)))))) )

(defun make-length-zero-lambda-before-refactor ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     (t (1+ (funcall length-lambda (cdr l)))))) )
     (function eternity)) )

(defun make-length-zero-lambda-refactored ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall make-length-lambda
          (function eternity)))
     (lambda (some-length-function)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     (t (1+ (funcall some-length-function (cdr l)))))) ) ) )

(defun make-length-zero-lambda ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
```

```
       (funcall make-length-lambda
          make-length-lambda )) ;here we don't need to get
              ;the function because
              ;make-length-lambda is
              ;already the function to
              ;use
     (lambda (make-length-lambda)     ;using this name for the
              ;argument we have a
              ;remainder that the first
              ;argument of
              ;make-length-lambda
              ;(because this is what
              ;this lambda is!) is
              ;make-length-lambda
              ;itself!
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     ;; the following invocation (funcall
     ;; make-length-lambda (cdr l)) doesn't work because
     ;; make-length-lambda expect a lambda as argument,
     ;; while here we provide (cdr l): the only case that
     ;; it works is that (cdr l) is a lambda but this is
     ;; impossible because if l is a non-empty list, (cdr
     ;; l) is a list, by the Law of Cdr!
     (t (1+ (funcall make-length-lambda (cdr l)))))) ) ) )

(defun make-length-but-not-seems-like-length ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall make-length-lambda
          make-length-lambda ))
     (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     (t (1+ (funcall
       ;; TLS: "Could we do this more than once?
       ;; Yes, just keep passing make-length-lambda
       ;; to itself, and we can do this as often as
       ;; we need to! How does it work? It keeps
       ;; adding recursive uses by passing
       ;; make-length-lambda to itself, just as it
       ;; is about to expire."  Me: with this
       ;; invocation we only build a new layer of
       ;; the recursion tower, because this
       ;; invocation return a lambda and doing this,
```

```
       ;; it doesn't make call like this, so this
       ;; invocation always halts.
       (funcall make-length-lambda
          make-length-lambda)
       (cdr l)))))) ) ) )

(defun make-length-toward-soundness ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall make-length-lambda
          make-length-lambda ))
     (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall
        (lambda (length)
    (lambda (l)
      (cond
        ((null l) 0)
        (t (1+ (funcall length (cdr l)))))) )
        (lambda (x)    ;usign this extract-pattern we
          ;don't invoke
          ;make-length-lambda with
          ;argument itself directly as
          ;the wrong before versione,
          ;but we build only a lambda
          ;that potentially does that
          ;invocation
    (funcall (funcall make-length-lambda ;from this
                 ;funcall returns
                 ;a lambda, by
                 ;contract of
                 ;make-length-lambda
          make-length-lambda)
       x))) )))

(defun make-length-toward-y-combinator ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall
   (lambda (le)
     (funcall (lambda (make-length-lambda)
    (funcall make-length-lambda
       make-length-lambda ))
        (lambda (make-length-lambda)
    (funcall le
       (lambda (x)
         (funcall
          (funcall make-length-lambda
             make-length-lambda) x))) )))
   (lambda (length)
     (lambda (l)
```

```
        (cond
    ((null l) 0)
    (t (1+ (funcall length (cdr l)))))))))

(defun y-combinator (le)
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (l)
        (funcall l l ))
      (lambda (l)
        (funcall le
           (lambda (x)
       (funcall (funcall l l) x))) )))

(defun length-y-combinator-powered (l)
  (funcall (y-combinator
       (lambda (length)
         (lambda (l)
     (cond
       ((null l) 0)
       (t (1+ (funcall length (cdr l)))))))) l))



(defun make-length-doesnt-halts ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
        (funcall make-length-lambda
           make-length-lambda ))
      (lambda (make-length-lambda)
        (funcall
         (lambda (length)
     (lambda (l)
       (cond
         ((null l) 0)
         (t (1+ (funcall length (cdr l)))))))
         (funcall make-length-lambda ;written in this way we have
             ;an infinite calls due to
             ;this funcall
            make-length-lambda)))))



(defun make-length-at-most-one-with-mklength ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
        "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
        (funcall make-length-lambda
           make-length-lambda )) ;here we don't need to get
               ;the function because
               ;make-length-lambda is
               ;already the function to
```

```
              ;use
      (lambda (make-length-lambda)      ;using this name for the
              ;argument we have a
              ;remainder that the first
              ;argument of
              ;make-length-lambda
              ;(because this is what
              ;this lambda is!) is
              ;make-length-lambda
              ;itself!
        "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
        (lambda (l)
          (cond
      ((null l) 0)
      (t (1+ (funcall
        (funcall make-length-lambda
           ;; fixing this function we build a
           ;; lambda that at the second
           ;; invocation build another lambda
           ;; that has the ETERNITY function
           ;; hardcoded, so the entire
           ;; make-length-at-most-one-with-mklength
           ;; will handle only list of at most
           ;; one argument
           (function eternity))
        (cdr l))))))) ) )

(defun make-length-at-most-one-lambda-before-refactor ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall
   (lambda (length-lambda)
     "contract: (lambda: list-of-sexp -> number) -> (lambda:
         list-of-sexp
  -> number)"
     (lambda (l)
       (cond
   ((null l) 0)
   (t (1+ (funcall length-lambda (cdr l)))))) )
   (funcall        ;the result of this function
           ;call is passed as argument to
           ;the above function call
    (lambda (length-lambda)
      "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
      (lambda (l)
  (cond
    ((null l) 0)
    (t (1+ (funcall length-lambda (cdr l)))))))
    (function eternity)      ;this function is passed as
           ;argument to the strictly
           ;above lambda
    ) ))
```

```
(defun make-length-at-most-one-lambda ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall make-length-lambda
          (funcall make-length-lambda
             (function eternity))))
     (lambda (some-length-function)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     (t (1+ (funcall some-length-function (cdr l)))))) ) ) )

(defun make-length-at-most-two-lambda-before-refactor ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall
   (lambda (length-lambda)
     "contract: (lambda: list-of-sexp -> number) -> (lambda:
         list-of-sexp
  -> number)"
     (lambda (l)
       (cond
   ((null l) 0)
   (t (1+ (funcall length-lambda (cdr l)))))) )
   (funcall
    (lambda (length-lambda)
      "contract: (lambda: list-of-sexp -> number) -> (lambda:
          list-of-sexp
  -> number)"
      (lambda (l)
  (cond
    ((null l) 0)
    (t (1+ (funcall length-lambda (cdr l)))))) )
    (funcall        ;the result of this function
         ;call is passed as argument to
         ;the above function call
     (lambda (length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
   (cond
     ((null l) 0)
     (t (1+ (funcall length-lambda (cdr l)))))))
     (function eternity)      ;this function is passed as
         ;argument to the strictly
         ;above lambda
     ) )))

(defun make-length-at-most-two-lambda ()
```

```
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall make-length-lambda
          (funcall make-length-lambda
             (funcall make-length-lambda
          (function eternity)))))
     (lambda (some-length-function)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     (t (1+ (funcall some-length-function (cdr l)))))) ) ) )

(defun make-length-at-most-three-lambda ()
  "contract: -> (lambda: list-of-sexp -> number)"
  (funcall (lambda (make-length-lambda)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (funcall make-length-lambda
          (funcall make-length-lambda
             (funcall make-length-lambda
          (funcall make-length-lambda
             (function
             eternity))))))
     (lambda (some-length-function)
       "contract: (lambda: list-of-sexp -> number) -> (lambda:
list-of-sexp -> number)"
       (lambda (l)
         (cond
     ((null l) 0)
     (t (1+ (funcall some-length-function (cdr l)))))) ) ) )

(defun new-entry (first second)
  "contract: list-of-sexp list-of-sexp -> pair"
  (build-pair first second))

(defun lookup-in-entry (name entry failure-lambda)
  "contract: atom pair (lambda: atom -> object) -> sexp"
  (lookup-in-entry-helper name
        (pair-first-component entry)
        (pair-second-component entry)
        failure-lambda))

(defun lookup-in-entry-helper (name names values failure-lambda)
  "contract: atom list-of-sexp list-of-sexp (lambda: atom -> object)
  -> sexp"
  (cond
    ((null names) (funcall failure-lambda name))
    ((eq name (car names)) (car values))
    (t (lookup-in-entry-helper name (cdr names) (cdr values)
```

```lisp
    failure-lambda)) ) )

(defun extend-table (entry table)
  "contract: entry table -> table"
  (cons entry table))

(defun lookup-in-table (name table failure-lambda)
  "contract: atom list-of-entry (lambda: atom -> object) -> atom"
  (cond
    ((null table) (funcall failure-lambda name))
    (t (lookup-in-entry name (car table)
      (lambda (missing-name)
        (lookup-in-table missing-name ;just to
              ;eliminate a
              ;style
              ;warning, we
              ;could have
              ;used NAME
              ;instead, the
              ;behaviour
              ;isn't
              ;affected
             (cdr table)
             failure-lambda))))))

(defun expression-to-action (e)
  "contract: sexp -> (lambda: sexp table -> sexp)"
  (cond
    ((atomp e) (atom-to-action e))
    (t (list-to-action e)) ))

(defun atom-to-action (e)
  "contract: atom -> (lambda: sexp table -> sexp)"
  (cond
    ((numberp e) (function *const))
    ((eq e (quote t)) (function *const))
    ((eq e (quote ())) (function *const))
    ((eq e (quote cons)) (function *const))
    ((eq e (quote car)) (function *const))
    ((eq e (quote cdr)) (function *const))
    ((eq e (quote null)) (function *const))
    ((eq e (quote equal-sexps)) (function *const))
    ((eq e (quote atomp)) (function *const))
    ((eq e (quote zerop)) (function *const))
    ((eq e (quote 1+)) (function *const))
    ((eq e (quote 1-)) (function *const))
    ((eq e (quote numberp)) (function *const))
    (t (function *identifier)) ) )

(defun list-to-action (e)
  "contract: list-of-sexp -> (lambda: sexp table -> sexp)"
  (cond
    ((atomp (car e))
```

```
      (cond
        ((eq (car e) (quote quote)) (function *quote))
        ((eq (car e) (quote lambda)) (function *lambda))
        ((eq (car e) (quote cond)) (function *cond))
        (t (function *application))))
    (t (function *application)) ))

(defun tls-eval (e)
  "contract: sexp -> sexp"
  (meaning e (quote ())))

(defun meaning (e table)
  "contract: sexp table -> sexp"
  (funcall (expression-to-action e) e table) )

(defun *const (e table)
  "contract: sexp table -> sexp"
  (cond
    ((numberp e) e)      ;a number evaluates to its
          ;value itself
    ((eq e (quote t)) t)
    ((eq e (quote ())) (quote ()))
    (t (build-pair (quote primitive) e)) ) )

(defun *quote (e table)
  "contract: sexp table -> sexp"
  (text-of e) )

(defun text-of (e)
  "contract: list-of-sexp (pair) -> list-of-sexp (pair)"
  (pair-second-component e) )

(defun *identifier (e table)
  "contract: sexp table -> sexp"
  (lookup-in-table e table (lambda (missing-name)
           (car (quote ())))) ) ;this is another way
              ;to raise an
              ;exception

(defun *lambda (e table)
  "contract: sexp table -> sexp"
  (build-pair (quote non-primitive)
        (cons table (cdr e))) ) ;the (cdr e) is a list that
          ;contains the formal
          ;parameters of the lambda
          ;expression and the lambda's
          ;body

(defun table-of (e)
  "contract: list-of-sexp -> sexp"
  (pair-first-component e) )

(defun formals-of (e)
```

```
  "contract: list-of-sexp -> sexp"
  (pair-second-component e) )

(defun body-of (e)
  "contract: list-of-sexp -> sexp"
  (pair-third-component e) )

(defun evcon (lines table)
  (cond
    ((elsep (question-of (car lines)))
     (meaning (answer-of (car lines)) table))
    ((meaning (question-of (car lines)) table)
     (meaning (answer-of (car lines)) table))
    (t (evcon (cdr lines) table))))

(defun elsep (x)
  "contract: list-of-sexp -> boolean"
  (cond
    ((atomp x) (eq x (quote t)))
    (t (quote ())) ))

(defun question-of (line)
  "contract: pair -> sexp"
  (pair-first-component line))

(defun answer-of (line)
  "contract: pair -> sexp"
  (pair-second-component line))

(defun *cond (e table)
  (evcon (cond-lines-of e) table))

(defun cond-lines-of (e)
  (cdr e))

(defun evlist (args table)
  (cond
    ((null args) (quote ()))
    (t (cons (meaning (car args) table)
       (evlist (cdr args) table)))))

(defun *application (e table)
  (tls-apply (meaning (function-of e) table)
       (evlist (arguments-of e) table)) )

(defun function-of (e)
  (car e))

(defun arguments-of (e)
  (cdr e))

(defun tls-apply (fun-representation evaluated-arguments)
  (cond
```

```
    ((primitivep fun-representation)
     (apply-primitive (second fun-representation)
         evaluated-arguments))
    ((non-primitivep fun-representation)
     (apply-closure (second fun-representation)
       evaluated-arguments))))

(defun primitivep (fun-representation)
  (eq (car fun-representation) (quote primitive)))

(defun non-primitivep (fun-representation)
  (eq (car fun-representation) (quote non-primitive)))

(defun apply-primitive (name vals)
  (cond
    ((eq name (quote cons)) (cons (first vals) (second vals)))
    ((eq name (quote car)) (car (first vals)))
    ((eq name (quote cdr)) (cdr (first vals)))
    ((eq name (quote null)) (null (first vals)))
    ((eq name (quote equal-sexps)) (equal-sexps (first vals) (second
       vals)))
    ((eq name (quote atomp)) (atomp-for-apply-primitive (first vals)))
    ((eq name (quote zerop)) (zerop (first vals)))
    ((eq name (quote 1+)) (1+ (first vals)))
    ((eq name (quote 1-)) (1- (first vals)))
    ((eq name (quote numberp)) (numberp (first vals))) ))

(defun atomp-for-apply-primitive (x)
  (cond
    ((atomp x) t)
    ((null x) (quote ()))
    ((eq (car x) (quote primitive)) t)
    ((eq (car x) (quote non-primitive)) t)
    (t nil)))

(defun apply-closure (closure vals)
  (meaning (body-of closure)
     (extend-table (new-entry (formals-of closure) vals)
           (table-of closure))))
```

1.2 JAVA IMPLEMENTATION OF Y-COMBINATOR

This implementation may seem repetitive and not object oriented: I've choosen to do it in this way because every step is immediately available without resorting to Git history.

```
/**
 * This interface allow us to stick to the Ninth Commandment, in
    particular it
 * allow us to abstract the common similarities.<br>
 * <br>
```

```
 * What does that mean? In order to "abstract the common similarities"
   we have
 * always to define an high order object, such that keep as argument
   what
 * changes (as form of a function) and return a function which
   encapsulate the
 * common behavior. The returned function do curry-ing on the passed
   argument,
 * so that the high-order object is a factory of functions.<br>
 * <br>
 * To be a little more formal this interface define the responsibility
   that an
 * implementor should have to be used as an high order object: it have
   to take a
 * function and return a function with the same signature (and semantic
   ).
 *
 * @author massimo
 *
 */
public interface ListLengthCalculatorHighOrder {

  ListLengthCalculator combine(ListLengthCalculator calculator);
}
```

```
/**
 * In order to remove the duplication given by the repetitive
   definition of the
 * "common similarities" part of high-order object, we introduce a
   little
 * "algebra-style" manipulation, that is we introduce an object that
   takes the
 * high-order object and returns what the high-order object returns.
 *
 * @author massimo
 *
 */
public interface ListLengthCalculatorMaker {
  ListLengthCalculator map(ListLengthCalculatorHighOrder highOrder);
}
```

```
public interface ListLengthCalculatorRecursiveInvocation {
  ListLengthCalculator invokeWithSelfRecursion(
      ListLengthCalculatorRecursiveInvocation self);
}
```

```

public interface ListLengthCalculator {
  int length(ListModule list);
}
```

```
/**
 * @author massimo
 *
 */
public class ListLengthCalculators {

  /**
   * This is a decider, always halts.
   */
  public static ListLengthCalculator NormalRecursion = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {
      return list.size() == 0 ? 0 : 1 + this.length(list.cdr());
    }
  };

  /**
   * This is the most partial function that we can write: it doesn't
       halt on
   * any input list.
   */
  public static ListLengthCalculator Eternity = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {
      return this.length(list);
    }
  };

  public static ListLengthCalculator DecideOnlyEmptyLists = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {
      return list.size() == 0 ? 0 : 1 + Eternity.length(list.cdr());
    }
  };

  public static ListLengthCalculator DecideOnlyListWithAtMostOneElement
       = new ListLengthCalculator() {

    @Override
    public int length(ListModule list) {
      return list.size() == 0 ? 0 : 1 + (new ListLengthCalculator() {

        @Override
        public int length(ListModule list) {
          return list.size() == 0 ? 0 : 1 + Eternity.length(list
              .cdr());
```

```java
        }
      }).length(list.cdr());

    }
  };

  public static ListLengthCalculator
      DecideOnlyListWithAtMostTwoElements = new ListLengthCalculator()
      {

    @Override
    public int length(ListModule list) {

      // in this definition we recognize a common structure: we ask
      // something about the size of the list and then delegate to a
      // function (constructed dynamically) to do so work. This
          function
      // asks the same question about the size of the list and then
      // delegate to a new function...
      return list.size() == 0 ? 0 : 1 + (new ListLengthCalculator() {
        @Override
        public int length(ListModule list) { // label:*

          return list.size() == 0 ? 0
              : 1 + (new ListLengthCalculator() {

                @Override
                public int length(ListModule list) { // label:**

                  // observe that here we fix the finish of
                  // the chain because we hard-code the use of
                  // Eternity. In this way if we want to
                  // extend the chain, we're forced to make a
                  // copy of the same code because this is a
                  // fixed structure (is something like
                  // enumerating all our knowledge)
                  return list.size() == 0 ? 0 : 1 + Eternity
                      .length(list.cdr());
                }
              }).length(list.cdr()); // this invocation executes
                          // label:**
        }
      }).length(list.cdr()); // this invocation executes label:*

    }
  };

  public static ListLengthCalculator DecideOnlyEmptyListsUsingHighOrder
      = new ListLengthCalculator() {

    @Override
    public int length(ListModule list) {
```

```
      // for extracting the similarities saw in the previous
          calculators,
      // we follow the Ninth Commandment, introducing the
      // ListLengthCalculatorHighOrder interface
      return (new ListLengthCalculatorHighOrder() {

        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).combine(Eternity).length(list);

    }
  };

  public static ListLengthCalculator
      DecideOnlyListWithAtMostOneElementUsingHighOrder = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      return (new ListLengthCalculatorHighOrder() {

        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).combine((new ListLengthCalculatorHighOrder() {

        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {
```

```
          @Override
          public int length(ListModule list) {
            return list.size() == 0 ? 0 : 1 + calculator
                .length(list.cdr());
          }
        };
      }
    }).combine(Eternity)).length(list);

  }
};

public static ListLengthCalculator
    DecideOnlyListWithAtMostTwoElementUsingHighOrder = new
    ListLengthCalculator() {

  @Override
  public int length(ListModule list) {

    // here we see the devastating effect of having introduced the
        high
    // order object: the structure of the code is the same as before
    // with the same nested levels, with the difference that, using
        the
    // high-order object we have the SAME code like a truly
        duplication
    // (before we have the same PATTERN, not the same CODE). What is
    // duplicated is the "common similarities" part, but couldn't be
    // otherwise because that part is the PATTERN in common in the
    // previous code. Now we can observe that the creation of the
    // high-order object isn't necessary to be replicated, but we can
    // use only one of them. This is of course a more truly
        duplication,
    // hence we are toward another refactoring...
    return (new ListLengthCalculatorHighOrder() {

      @Override
      public ListLengthCalculator combine(
          final ListLengthCalculator calculator) {

        return new ListLengthCalculator() {

          @Override
          public int length(ListModule list) {
            return list.size() == 0 ? 0 : 1 + calculator
                .length(list.cdr());
          }
        };
      }
    }).combine((new ListLengthCalculatorHighOrder() {

      @Override
```

```
      public ListLengthCalculator combine(
          final ListLengthCalculator calculator) {

        return new ListLengthCalculator() {

          @Override
          public int length(ListModule list) {
            return list.size() == 0 ? 0 : 1 + calculator
                .length(list.cdr());
          }
        };
      }
    }).combine((new ListLengthCalculatorHighOrder() {

      @Override
      public ListLengthCalculator combine(
          final ListLengthCalculator calculator) {

        return new ListLengthCalculator() {

          @Override
          public int length(ListModule list) {
            return list.size() == 0 ? 0 : 1 + calculator
                .length(list.cdr());
          }
        };
      }
    }).combine(Eternity))).length(list);

  }
};

public static ListLengthCalculator
    DecideOnlyEmptyListsUsingHighOrder_mklength = new
    ListLengthCalculator() {

  @Override
  public int length(ListModule list) {

    return (new ListLengthCalculatorMaker() {

      @Override
      public ListLengthCalculator map(
          ListLengthCalculatorHighOrder highOrder) {

        return highOrder.combine(Eternity);
      }
    }).map(new ListLengthCalculatorHighOrder() {

      @Override
      public ListLengthCalculator combine(
          final ListLengthCalculator calculator) {
```

```
          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).length(list);

    }
  };

  public static ListLengthCalculator
      DecideOnlyListWithAtMostOneElementUsingHighOrder_mklength = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      return (new ListLengthCalculatorMaker() {

        @Override
        public ListLengthCalculator map(
            ListLengthCalculatorHighOrder highOrder) {

          return highOrder.combine(highOrder.combine(Eternity));
        }
      }).map(new ListLengthCalculatorHighOrder() {

        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).length(list);

    }
  };

  public static ListLengthCalculator
      DecideOnlyListWithAtMostTwoElementUsingHighOrder_mklength = new
      ListLengthCalculator() {
```

```
    @Override
    public int length(ListModule list) {

      // using this algebra-style manipulation, instead of building and
      // invoking three times the high-order object, we build it only once
      // and pass the object to a function that invoke it three times. In
      // this way we divide when we build the object from when we use it.
      // Observe that the nested structure of the invocations of the
      // high-order object is preserved during this transformation.
      return (new ListLengthCalculatorMaker() {

        @Override
        public ListLengthCalculator map(
            ListLengthCalculatorHighOrder highOrder) {

          return highOrder.combine(highOrder.combine(highOrder
              .combine(Eternity)));
        }
      }).map(new ListLengthCalculatorHighOrder() {

        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).length(list);

    }
  };

  public static ListLengthCalculator
      DecideOnlyListWithAtMostThreeElementUsingHighOrder_mklength = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      return (new ListLengthCalculatorMaker() {

        @Override
        public ListLengthCalculator map(
            ListLengthCalculatorHighOrder highOrder) {
```

```
          return highOrder.combine(highOrder.combine(highOrder
              .combine(highOrder.combine(Eternity))));
        }
      }).map(new ListLengthCalculatorHighOrder() {

        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).length(list);

    }
  };

  /**
   * First version equivalent to the normal length implementation.
   */
  public static ListLengthCalculator DecideListLengthUsingSelfRecursion
      = new ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      // introducing the self invocation allow us to add a little brick
      // every time we need in order to match the length of the list.
      return (new ListLengthCalculatorRecursiveInvocation() {

        @Override
        public ListLengthCalculator invokeWithSelfRecursion( // label:*
            ListLengthCalculatorRecursiveInvocation self) {

          return self.invokeWithSelfRecursion(self); // invokes
                                    // label:**

        }
      }).invokeWithSelfRecursion( // invokes label:*
          new ListLengthCalculatorRecursiveInvocation() {

            // using the self recursion instead of the high-order
            // object we loose the syntactic equivalence of the code
            // that compute the length. Instead of using the
            // calculator as done in the previous version here we
            // use the self invocation
```

```
            // self.invokeWithSelfRecursion(self) which build a
            // calculator, but this is far different than using a
            // collector available from the high-order object.

            @Override
            public ListLengthCalculator invokeWithSelfRecursion( //
                label:**
                final ListLengthCalculatorRecursiveInvocation self) {

              return new ListLengthCalculator() {

                @Override
                public int length(ListModule list) {

                  // the invocation
                  // self.invokeWithSelfRecursion(self)
                  // doesn't produce a loop because it build
                  // an object of type ListLengthCalculator,
                  // without calling the length method (this
                  // method!) which, if invoked, will
                  // produce a loop instead.
                  return list.size() == 0 ? 0 : 1 + self
                      .invokeWithSelfRecursion(self)
                      .length(list.cdr());
                }
              };
            }
          }).length(list);

    }
  };

  public static ListLengthCalculator
      UndecideListLengthExtractingSelfRecursion = new
      ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      return (new ListLengthCalculatorRecursiveInvocation() {

        @Override
        public ListLengthCalculator invokeWithSelfRecursion(
            ListLengthCalculatorRecursiveInvocation self) {

          return self.invokeWithSelfRecursion(self);

        }
      }).invokeWithSelfRecursion(
          new ListLengthCalculatorRecursiveInvocation() {

            @Override
            public ListLengthCalculator invokeWithSelfRecursion(
```

```
              final ListLengthCalculatorRecursiveInvocation self) {

            // in this way we return back using the high-order
            // object and this structure is the same as the
            // firsts, where the code that asks on the lists use
            // the passed calculator. Here we've used the same
            // algebra-style manipulation: build the necessary
            // object (in this case the construction is done in
            // the invocation
            // self.invokeWithSelfRecursion(self)) pass it to an
            // high-order object and from the inside invoke it.
            // This produce an error not in this algebra
            // manipulation, but in the self invocation which
            // produce an infinite loop.
            return (new ListLengthCalculatorHighOrder() {

              @Override
              public ListLengthCalculator combine(
                  final ListLengthCalculator calculator) {

                return new ListLengthCalculator() {

                  @Override
                  public int length(ListModule list) {

                    return list.size() == 0 ? 0
                        : 1 + calculator
                            .length(list.cdr());

                  }
                };

              }
            }).combine(self.invokeWithSelfRecursion(self));

          }
        }).length(list);

  }
};

public static ListLengthCalculator
    DecideListLengthUsingSelfRecursionTowardYCombinatorFirstStep =
    new ListLengthCalculator() {

  @Override
  public int length(ListModule list) {

    return (new ListLengthCalculatorRecursiveInvocation() {

      @Override
      public ListLengthCalculator invokeWithSelfRecursion(
          ListLengthCalculatorRecursiveInvocation self) {
```

```java
          return self.invokeWithSelfRecursion(self);

        }
      }).invokeWithSelfRecursion(
          new ListLengthCalculatorRecursiveInvocation() {

            // in this version we haven't the high-order object, but
            // we make a little step toward its re-introduction

            @Override
            public ListLengthCalculator invokeWithSelfRecursion(
                final ListLengthCalculatorRecursiveInvocation self) {

              return new ListLengthCalculator() {

                @Override
                public int length(ListModule list) {
                  return list.size() == 0 ? 0 : 1 +
                  // building here a calculator object and
                  // invoking it allow us to "reintroduce" the
                  // concept of a calculator (like the first
                  // versions) which hides the use of the self
                  // recursion.
                      (new ListLengthCalculator() {

                        @Override
                        public int length(
                            ListModule list) {
                          return self
                              .invokeWithSelfRecursion(
                                  self)
                              .length(list);
                        }
                      }).length(list.cdr());
                }
              };
            }
          }).length(list);

    }
  };

  public static ListLengthCalculator
      DecideListLengthUsingSelfRecursionTowardYCombinatorSecondStep =
      new ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      return (new ListLengthCalculatorRecursiveInvocation() {

        @Override
```

```
  public ListLengthCalculator invokeWithSelfRecursion(
      ListLengthCalculatorRecursiveInvocation self) {

    return self.invokeWithSelfRecursion(self);

  }
}).invokeWithSelfRecursion(
    new ListLengthCalculatorRecursiveInvocation() {

      @Override
      public ListLengthCalculator invokeWithSelfRecursion(
          final ListLengthCalculatorRecursiveInvocation self) {

        // here we reintroduce the high-order object just
        // with the same algebra manipulation: the moral
        // behind this re-introduction is that when we are
        // building an object dynamically, don't define it
        // where needed and use it immediately, instead
        // build it and pass it to an high-order object
        // which in turn will invoke it. We observe that
        // doing this, the object self is no more used in
        // the definition of the high-order object, hence we
        // have another refactor to do! (before, where we
        // build dynamically the calculator directly, we
        // have the dependency from self in the compound
        // where we did the definition.
        return (new ListLengthCalculatorHighOrder() {

          @Override
          public ListLengthCalculator combine(
              final ListLengthCalculator calculator) {

            return new ListLengthCalculator() {

              @Override
              public int length(ListModule list) {
                return list.size() == 0 ? 0
                    : 1 + calculator
                        .length(list.cdr());
              }
            };
          }
        }).combine(new ListLengthCalculator() {

          @Override
          public int length(ListModule list) {
            return self.invokeWithSelfRecursion(self)
                .length(list);
          }
        });
      }
    }).length(list);
```

```java
    }
  };

  public static ListLengthCalculator
      DecideListLengthUsingSelfRecursionTowardYCombinatorThirdStep =
      new ListLengthCalculator() {

    @Override
    public int length(ListModule list) {

      // another application of the algebra-style manipulation: build the
      // high-order object outside the place where you need it, pass it to
      // another high-order object and then use it through its reference
      // parameter.
      return (new ListLengthCalculatorMaker() {

        @Override
        public ListLengthCalculator map(
            final ListLengthCalculatorHighOrder highOrder) {

          return (new ListLengthCalculatorRecursiveInvocation() {

            @Override
            public ListLengthCalculator invokeWithSelfRecursion(
                ListLengthCalculatorRecursiveInvocation self) {

              return self.invokeWithSelfRecursion(self);

            }
          }).invokeWithSelfRecursion(new
              ListLengthCalculatorRecursiveInvocation() {

            @Override
            public ListLengthCalculator invokeWithSelfRecursion(
                final ListLengthCalculatorRecursiveInvocation self) {

              return highOrder
                  .combine(new ListLengthCalculator() {

                    @Override
                    public int length(ListModule list) {
                      return self
                          .invokeWithSelfRecursion(
                              self).length(list);
                    }
                  });
            }
          });
        }
      }).map(new ListLengthCalculatorHighOrder() {
```

```
        @Override
        public ListLengthCalculator combine(
            final ListLengthCalculator calculator) {

          return new ListLengthCalculator() {

            @Override
            public int length(ListModule list) {
              return list.size() == 0 ? 0 : 1 + calculator
                  .length(list.cdr());
            }
          };
        }
      }).length(list);

    }
  };

}
```

```
import java.util.LinkedList;
import java.util.List;

public class ListModule {
  private List<Object> internal_list = new LinkedList<Object>();

  private ListModule(List<Object> list) {
    internal_list = new LinkedList<Object>(list);
  }

  public Object car() {
    return internal_list.get(0);
  }

  public ListModule cdr() {
    return new ListModule(internal_list.subList(1, internal_list.size()
        ));
  }

  public ListModule cons(Object obj) {
    ListModule new_list = new ListModule(internal_list);
    new_list.internal_list.add(0, obj);
    return new_list;
  }

  public static ListModule nil() {
    return new ListModule(new LinkedList<Object>());
  }

  public int size() {
    return internal_list.size();
```

```java
  }

  public static ListModule makeWithRequestedCardinality(int cardinality
     ) {
    ListModule result = ListModule.nil();

    for (int counter = 0; counter < cardinality; counter = counter + 1)
        {

      result = result.cons(new Object());
    }

    return result;

  }
}
```

```java
package Ycombinator;

public class Ycombinator<Input, Output> {

  public interface InterfaceType<InterfaceInput, InterfaceOutput> {
    InterfaceOutput compute(InterfaceInput input);
  }

  public interface HighOrderCombinatorFor<Interface> {
    Interface combine(Interface calculator);
  }

  private interface RecursiveInvocationFor<Interface> {
    Interface invokeWithSelfRecursion(RecursiveInvocationFor<Interface>
        self);
  }

  private interface FromHighOrderCombinatorTo<Interface> {
    Interface map(HighOrderCombinatorFor<Interface> highOrder);
  }

  public InterfaceType<Input, Output> recursion(
      HighOrderCombinatorFor<InterfaceType<Input, Output>>
          highOrderObject) {

    return (new FromHighOrderCombinatorTo<InterfaceType<Input, Output
        >>() {

      @Override
      public InterfaceType<Input, Output> map(
          final HighOrderCombinatorFor<InterfaceType<Input, Output>>
              highOrder) {

        return (new RecursiveInvocationFor<InterfaceType<Input, Output
            >>() {
```

```java
          @Override
          public InterfaceType<Input, Output> invokeWithSelfRecursion(
              RecursiveInvocationFor<InterfaceType<Input, Output>> self
                  ) {

            return self.invokeWithSelfRecursion(self);

          }
        }).invokeWithSelfRecursion(new RecursiveInvocationFor<
            InterfaceType<Input, Output>>() {

          @Override
          public InterfaceType<Input, Output> invokeWithSelfRecursion(
              final RecursiveInvocationFor<InterfaceType<Input, Output
                  >> self) {

            return highOrder
                .combine(new InterfaceType<Input, Output>() {

                  @Override
                  public Output compute(Input input) {
                    return self.invokeWithSelfRecursion(self)
                        .compute(input);
                  }
                });
          }
        });
      }
    }).map(highOrderObject);
  }
}
```

```java
import org.junit.Assert;
import org.junit.Test;

import Ycombinator.Ycombinator;
import Ycombinator.Ycombinator.HighOrderCombinatorFor;
import Ycombinator.Ycombinator.InterfaceType;

public class Unittests {

  private static ListModule empty_list = ListModule.nil();
  private static ListModule list_with_one_element = ListModule.nil().
      cons(
      new Object());
  private static ListModule list_with_two_elements = ListModule.nil()
      .cons(new Object()).cons(new Object());
  private static ListModule list_with_three_elements = ListModule.nil()
      .cons(new Object()).cons(new Object()).cons(new Object());
  private static ListModule list_with_four_elements = ListModule.nil()
      .cons(new Object()).cons(new Object()).cons(new Object())
```

```
    .cons(new Object());
private static ListModule list_with_five_elements = ListModule.nil()
    .cons(new Object()).cons(new Object()).cons(new Object())
    .cons(new Object()).cons(new Object());

@Test
public void checking_lists_for_tests_execution() {

  Assert.assertEquals(0, empty_list.size());
  Assert.assertEquals(1, list_with_one_element.size());
  Assert.assertEquals(2, list_with_two_elements.size());
  Assert.assertEquals(3, list_with_three_elements.size());
  Assert.assertEquals(4, list_with_four_elements.size());
  Assert.assertEquals(5, list_with_five_elements.size());
}

@Test
public void checking_lists_for_requested_cardinality() {

  int cardinality = 54;
  Assert.assertEquals(cardinality, ListModule
      .makeWithRequestedCardinality(cardinality).size());

}

@Test
public void default_list_length_calculator_should_compute_correctly()
     {

  ListLengthCalculator calculator = ListLengthCalculators.
      NormalRecursion;

  Assert.assertEquals(0, calculator.length(empty_list));

  Assert.assertEquals(3, calculator.length(list_with_three_elements))
      ;
  Assert.assertEquals(2, calculator.length(list_with_two_elements));
  Assert.assertEquals(1, calculator.length(list_with_one_element));
}

@Test(expected = StackOverflowError.class)
public void
    run_eternity_calculator_on_empty_list_should_produce_stack_overflow
    () {

  Assert.assertEquals(-1,
      ListLengthCalculators.Eternity.length(empty_list));
}

@Test
public void run_DecideOnlyEmptyLists_with_empty_list() {

  Assert.assertEquals(0,
```

```
        ListLengthCalculators.DecideOnlyEmptyLists.length(empty_list));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyEmptyLists_with_non_empty_list_should_produce_stack_overflow
    () {

  Assert.assertEquals(-1, ListLengthCalculators.DecideOnlyEmptyLists
        .length(list_with_one_element));
}

@Test
public void run_DecideOnlyListsWithAtMostOneElement_with_empty_list()
     {

  Assert.assertEquals(0,
        ListLengthCalculators.DecideOnlyListWithAtMostOneElement
            .length(empty_list));
}

@Test
public void
    run_DecideOnlyListsWithAtMostOneElement_with_one_element_in_list
    () {

  Assert.assertEquals(1,
        ListLengthCalculators.DecideOnlyListWithAtMostOneElement
            .length(list_with_one_element));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyListsWithAtMostOneElement_with_two_element_in_list_should_produce_s
    () {

  Assert.assertEquals(-1,
        ListLengthCalculators.DecideOnlyListWithAtMostOneElement
            .length(list_with_two_elements));
}

@Test
public void run_DecideOnlyListsWithAtMostTwoElement_with_empty_list()
     {

  Assert.assertEquals(0,
        ListLengthCalculators.DecideOnlyListWithAtMostTwoElements
            .length(empty_list));
}

@Test
```

```
public void
    run_DecideOnlyListsWithAtMostTwoElement_with_one_element_in_list
    () {

  Assert.assertEquals(1,
      ListLengthCalculators.DecideOnlyListWithAtMostTwoElements
          .length(list_with_one_element));
}

@Test
public void
    run_DecideOnlyListsWithAtMostTwoElement_with_two_element_in_list
    () {

  Assert.assertEquals(2,
      ListLengthCalculators.DecideOnlyListWithAtMostTwoElements
          .length(list_with_two_elements));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyListsWithAtMostTwoElement_with_two_element_in_list_should_produce_stack_ove
    () {

  Assert.assertEquals(-1,
      ListLengthCalculators.DecideOnlyListWithAtMostTwoElements
          .length(list_with_three_elements));
}

@Test
public void
    run_DecideOnlyEmptyLists_using_high_order_creation_with_empty_list
    () {

  Assert.assertEquals(0,
      ListLengthCalculators.DecideOnlyEmptyListsUsingHighOrder
          .length(empty_list));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyEmptyLists_using_high_order_creation_with_non_empty_list_should_produce_sta
    () {

  Assert.assertEquals(0,
      ListLengthCalculators.DecideOnlyEmptyListsUsingHighOrder
          .length(list_with_one_element));
}

@Test
public void
    run_DecideOnlyListsWithAtMostOneElement_using_high_order_creation_with_empty_list
    () {
```

```
    Assert.assertEquals(
        0,
        ListLengthCalculators.
            DecideOnlyListWithAtMostOneElementUsingHighOrder
            .length(empty_list));
}

@Test
public void
    run_DecideOnlyListsWithAtMostOneElement_using_high_order_creation_with_one_eleme
    () {

    Assert.assertEquals(
        1,
        ListLengthCalculators.
            DecideOnlyListWithAtMostOneElementUsingHighOrder
            .length(list_with_one_element));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyListsWithAtMostOneElement_using_high_order_creation_with_two_eleme
    () {

    Assert.assertEquals(
        -1,
        ListLengthCalculators.
            DecideOnlyListWithAtMostOneElementUsingHighOrder
            .length(list_with_two_elements));
}

@Test
public void
    run_DecideOnlyListsWithAtMostTwoElement_using_high_order_with_empty_list
    () {

    Assert.assertEquals(
        0,
        ListLengthCalculators.
            DecideOnlyListWithAtMostTwoElementUsingHighOrder
            .length(empty_list));
}

@Test
public void
    run_DecideOnlyListsWithAtMostTwoElement_using_high_order_with_one_element_in_list
    () {

    Assert.assertEquals(
        1,
        ListLengthCalculators.
            DecideOnlyListWithAtMostTwoElementUsingHighOrder
```

```
            .length(list_with_one_element));
}

@Test
public void
    run_DecideOnlyListsWithAtMostTwoElement_using_high_order_with_two_element_in_list
    () {

  Assert.assertEquals(
        2,
        ListLengthCalculators.
            DecideOnlyListWithAtMostTwoElementUsingHighOrder
            .length(list_with_two_elements));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyListsWithAtMostTwoElement_using_high_order_with_two_element_in_list_should_
    () {

  Assert.assertEquals(
        -1,
        ListLengthCalculators.
            DecideOnlyListWithAtMostTwoElementUsingHighOrder
            .length(list_with_three_elements));
}

@Test
public void
    run_DecideOnlyEmptyLists_using_mklength_creation_with_empty_list
    () {

  Assert.assertEquals(
        0,
        ListLengthCalculators.
            DecideOnlyEmptyListsUsingHighOrder_mklength
            .length(empty_list));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyEmptyLists_using_mklength_creation_with_non_empty_list_should_produce_stack
    () {

  Assert.assertEquals(
        0,
        ListLengthCalculators.
            DecideOnlyEmptyListsUsingHighOrder_mklength
            .length(list_with_one_element));
}

@Test
```

```
public void
    run_DecideOnlyListsWithAtMostOneElement_using_mklength_creation_with_empty_list
    () {

  Assert.assertEquals(
      0,
      ListLengthCalculators.
          DecideOnlyListWithAtMostOneElementUsingHighOrder_mklength
           .length(empty_list));
}

@Test
public void
    run_DecideOnlyListsWithAtMostOneElement_using_mklength_creation_with_one_element
    () {

  Assert.assertEquals(
      1,
      ListLengthCalculators.
          DecideOnlyListWithAtMostOneElementUsingHighOrder_mklength
           .length(list_with_one_element));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyListsWithAtMostOneElement_using_mklength_creation_with_two_element
    () {

  Assert.assertEquals(
      -1,
      ListLengthCalculators.
          DecideOnlyListWithAtMostOneElementUsingHighOrder_mklength
           .length(list_with_two_elements));
}

@Test
public void
    run_DecideOnlyListsWithAtMostTwoElement_using_mklength_with_empty_list
    () {

  Assert.assertEquals(
      0,
      ListLengthCalculators.
          DecideOnlyListWithAtMostTwoElementUsingHighOrder_mklength
           .length(empty_list));
}

@Test
public void
    run_DecideOnlyListsWithAtMostTwoElement_using_mklength_with_one_element_in_list
    () {

  Assert.assertEquals(
```

```
            1,
            ListLengthCalculators.
                DecideOnlyListWithAtMostTwoElementUsingHighOrder_mklength
                .length(list_with_one_element));
    }

    @Test
    public void
        run_DecideOnlyListsWithAtMostTwoElement_using_mklength_with_two_element_in_list
        () {

      Assert.assertEquals(
            2,
            ListLengthCalculators.
                DecideOnlyListWithAtMostTwoElementUsingHighOrder_mklength
                .length(list_with_two_elements));
    }

    @Test(expected = StackOverflowError.class)
    public void
        run_DecideOnlyListsWithAtMostTwoElement_using_mklength_with_two_element_in_list_should_pr
        () {

      Assert.assertEquals(
            -1,
            ListLengthCalculators.
                DecideOnlyListWithAtMostTwoElementUsingHighOrder_mklength
                .length(list_with_three_elements));
    }

    @Test
    public void
        run_DecideOnlyListsWithAtMostThreeElement_using_mklength_with_empty_list
        () {

      Assert.assertEquals(
            0,
            ListLengthCalculators.
                DecideOnlyListWithAtMostThreeElementUsingHighOrder_mklength
                .length(empty_list));
    }

    @Test
    public void
        run_DecideOnlyListsWithAtMostThreeElement_using_mklength_with_one_element_in_list
        () {

      Assert.assertEquals(
            1,
            ListLengthCalculators.
                DecideOnlyListWithAtMostThreeElementUsingHighOrder_mklength
                .length(list_with_one_element));
    }
```

```
@Test
public void
    run_DecideOnlyListsWithAtMostThreeElement_using_mklength_with_two_element_in_list
    () {

  Assert.assertEquals(
      2,
      ListLengthCalculators.
          DecideOnlyListWithAtMostThreeElementUsingHighOrder_mklength
           .length(list_with_two_elements));
}

@Test
public void
    run_DecideOnlyListsWithAtMostThreeElement_using_mklength_with_three_element_in_li
    () {

  Assert.assertEquals(
      3,
      ListLengthCalculators.
          DecideOnlyListWithAtMostThreeElementUsingHighOrder_mklength
           .length(list_with_three_elements));
}

@Test(expected = StackOverflowError.class)
public void
    run_DecideOnlyListsWithAtMostThreeElement_using_mklength_with_four_element_in_lis
    () {

  Assert.assertEquals(
      -1,
      ListLengthCalculators.
          DecideOnlyListWithAtMostThreeElementUsingHighOrder_mklength
           .length(list_with_four_elements));
}

@Test
public void run_DecideListLengthUsingSelfRecursion_with_all_list() {

  assert_correct_computation_for_all_testing_lists(
      ListLengthCalculators.DecideListLengthUsingSelfRecursion);

}

@Test
public void
    run_DecideListLengthUsingSelfRecursionTowardYCombinatorFirstStep_with_all_list
    () {

  assert_correct_computation_for_all_testing_lists(
      ListLengthCalculators.
      DecideListLengthUsingSelfRecursionTowardYCombinatorFirstStep);
```

```
}

@Test
public void
    run_DecideListLengthUsingSelfRecursionTowardYCombinatorSecondStep_with_all_list
    () {

  assert_correct_computation_for_all_testing_lists(
      ListLengthCalculators.
      DecideListLengthUsingSelfRecursionTowardYCombinatorSecondStep);

}

@Test
public void
    run_DecideListLengthUsingSelfRecursionTowardYCombinatorThirsStep_with_all_list
    () {

  assert_correct_computation_for_all_testing_lists(
      ListLengthCalculators.
      DecideListLengthUsingSelfRecursionTowardYCombinatorThirdStep);

}

private void assert_correct_computation_for_all_testing_lists(
    ListLengthCalculator calculator) {

  Assert.assertEquals(0, calculator.length(empty_list));

  Assert.assertEquals(1, calculator.length(list_with_one_element));

  Assert.assertEquals(2, calculator.length(list_with_two_elements));

  Assert.assertEquals(3, calculator.length(list_with_three_elements))
      ;

  Assert.assertEquals(4, calculator.length(list_with_four_elements));

  Assert.assertEquals(5, calculator.length(list_with_five_elements));

}

@Test(expected = StackOverflowError.class)
public void
    run_UndecideListLengthExtractingSelfRecursion_with_empty_list() {

  Assert.assertEquals(0,
      ListLengthCalculators.UndecideListLengthExtractingSelfRecursion
          .length(empty_list));
}

@Test
```

```
  public void YCombinator_examples() {
    Ycombinator<ListModule, Integer> y_combinator = new Ycombinator<
        ListModule, Integer>();

    InterfaceType<ListModule, Integer> entire_computation =
        y_combinator
        .recursion(new HighOrderCombinatorFor<Ycombinator.InterfaceType
            <ListModule, Integer>>() {

          @Override
          public InterfaceType<ListModule, Integer> combine(
              final InterfaceType<ListModule, Integer> calculator) {

            return new InterfaceType<ListModule, Integer>() {

              @Override
              public Integer compute(ListModule input) {

                return input.size() == 0 ? 0 : 1 + calculator
                    .compute(input.cdr());
              }
            };
          }
        });

    Assert.assertEquals((Integer) 0, entire_computation.compute(
        empty_list));

    Assert.assertEquals((Integer) 1,
        entire_computation.compute(list_with_one_element));

    Assert.assertEquals((Integer) 2,
        entire_computation.compute(list_with_two_elements));

    Assert.assertEquals((Integer) 3,
        entire_computation.compute(list_with_three_elements));

    Assert.assertEquals((Integer) 4,
        entire_computation.compute(list_with_four_elements));

    Assert.assertEquals((Integer) 5,
        entire_computation.compute(list_with_five_elements));

    Integer cardinality = 190;
    Assert.assertEquals(cardinality, entire_computation.compute(
        ListModule
        .makeWithRequestedCardinality(cardinality)));

  }
}
```