(24)  

I. **LISP Programming.** You cannot use PROGs or any LOOP/DO constructs.

An \( N \times N \) complex matrix \( R \) is represented by \( N \) lists of \( N \) pairs, where the first element of each pair is the real part of the matrix component and the second element of each pair is the imaginary part of the corresponding matrix component. Give a recursive LISP function or use a mapping function to compute the Hermitian, i.e., the conjugate transpose matrix \( R^\dagger \):

\[
\begin{pmatrix}
(1+j2) & (3+j4) \\
(5+j6) & (7+j8)
\end{pmatrix} \rightarrow \begin{pmatrix}
(1-j2) & (5-j6)^\dagger \\
(3-j4) & (7-j8)
\end{pmatrix}
\]

Lisp → `(CONJ-TRANSPOSE '((1 2) (3 4)) ((5 6) (7 8)))

`(((1 -2) (5 -6)) ((3 -4) (7 -8)))`

Lisp → `(CONJ-TRANSPOSE '((1 2) (3 4) (5 6)) ((7 8) (9 10) (11 12)) ((13 14) (15 16) (17 18)))`

`(((1 -2) (7 -8) (13 -14)) ((3 -4) (9 -10) (15 -16)) ((5 -6) (11 -12) (17 -18)))`

**Hint:** You may wish to write a recursive LISP function or use mapping functions to transpose an \( N \times N \) matrix represented as \( N \) lists of \( N \) integers each (and receive partial credit for your effort), e.g.,

Lisp → `(TRANSPOSE '((1 2 3) (4 5 6) (7 8 9)))`

`((1 4 7) (2 5 8) (3 6 9))`

Lisp → `(TRANSPOSE '((1 2) (3 4) (5 6)))`

`((1 3 5) (2 4 6))`

Lisp → `(CONJ-TRANSPOSE '((1 2) (3 4) (5 6)) ((7 8) (9 10) (11 12)) ((13 14) (15 16) (17 18)))`

`(((1 -2) (7 -8) (13 -14)) ((3 -4) (9 -10) (15 -16)) ((5 -6) (11 -12) (17 -18)))`

```lisp
(defun transp (mat) (if (null (car mat)) nil (cons (mapcar #'car mat) (transp (mapcar #'cdr mat)))))

(defun ins (l1 l2) (cond
 ( (null l1) nil)
 ( t (cons (cons (car l1) (car l2)) (ins (cdr l1) (cdr l2))))))

(defun negate (mat) (ins (mapcar #'car mat) (neg (mapcar #'cdr mat))))

(defun neg (x) (* -1 x))

(negate (mapcar #'car m))

((1 -2) (5 -6))

(defun ctransp (mat) (if (null (car mat)) nil (cons (negate (mapcar #'car mat)) (ctransp (mapcar #'cdr mat)))))

CTRANSP

> (CTRANSP M)

`(((1 -2) (5 -6)) ((3 -4) (7 -8)))`

> (CTRANSP '((1 2) (3 4) (5 6)) ((7 8) (9 10) (11 12)) ((13 14) (15 16) (17 18)))`

`(((1 -2) (7 -8) (13 -14)) ((3 -4) (9 -10) (15 -16)) ((5 -6) (11 -12) (17 -18)))`
```
II. LISP Programming. Choose between A or B below.

A. Write a function that inverts and consolidates an association list (a list of pairs or a list of dotted pairs, your choice!). For example,

LISP→(INVERT-ALIST '(McGriff . WR) (Jackson . RB) (Johnson . QB) (Gillespie . RB) (Taylor . WR) (Karim . WR))

((RB GILLESPIE JACKSON) (WR KARIM TAYLOR MCGRIFF) (QB JOHNSON))

LISP→(INVERT-ALIST '(Mc Griff WR) (Jackson RB) (Johnson QB) (Gillespie RB) (Taylor WR) (Karim WR))

((RB Gillespie Jackson) (WR Karim Taylor McGriff) (QB Johnson))

Suppose:

(setf X '(D E F))

Hint:

(rplacd X '(A B C)) → (D A B C)

or (setf (cdr X) '(A B C)) changes X to (D A B C)

B. Write a recursive function that finds the minimum number that occurs at any depth in a list. For example,

LISP→(MIN-NO ‘((A 3) (B (C 1.5)) 3.2))

1.5

(defun INVERT-ALIST (alist) (let ((newalist nil))
  (do ((entries alist (cdr entries))) ((null entries) newalist)
    (let ((oldkey (caar entries)) (oldvalue (cdar entries)))
      (let ((newentry (assoc oldvalue newalist)))
        (if (null newentry) (setq newalist (cons (list oldvalue oldkey) newalist))
         (rplacd newentry (cons oldkey (cdr newentry))))))))

(defun min-no (x) (helper x nil))
(defun helper(obj best) (cond
  ((numberp obj) (cond ((null obj) nil)
                       ((< best obj) best)
                       (t obj)))
  (atom obj) best)
  (t (min (helper (car obj) best) (helper (cdr obj) best))))
III. LISP Programming. Do both (a) and (b) below.

(12) a. MAPC is like MAPCAR in that it hands to its functional argument (funarg) successive cars of the input list. MAPC differs from MAPCAR in that MAPC prematurely stops applying the funarg to the successive cars of the input list if the value returned by the argument function is NIL. Further, MAPC returns NIL if it was stopped or T if it was allowed to exhaust the input list. This is useful because many of the functions we have defined in LISP should be stopped when the result is obvious and there is no need to traverse the input list until it is exhausted. Give a definition of MAPC.

(defun mapcar (f lis) (cond
  ( (null lis) nil )
  ( t (cons (funcall f (car lis)) (mapcar f (cdr lis)))) ))

(12) b. Define EQSET, a predicate that determines whether two sets are the same set using the MAPC mapping function.

Lisp> (eqset '(a b c) '(b a)) ==> Nil
Lisp> (eqset '(a b) '(a b c)) ==> Nil
Lisp> (eqset '(a b c) '(b a c)) ==> T
Lisp> (eqset '(coke is it) '(is it coke)) ==> T
Lisp> (eqset '(inhuman acts are human mistakes) '(human acts are inhuman mistakes)) ==> T

(defun MAPC (f Lis) (PROG (fvalue)
  LOOP
  ( when (null Lis) (return T) )
  ( setq fvalue (funcall f (car Lis)) )
  ( when (null fvalue) (return Nil) )
  ( setq Lis (cdr Lis) )
  ( go LOOP ) ))

alternatively

(defun mapc (f lis) (cond
  ((null lis) t)
  ( (funcall f (car lis)) (mapc f (cdr lis)))
  ( t nil )))

(defun EQSET (s1 s2) (cond
  ((null s1) (null s2))
  ((null s2) nil)
  ( (eq (length s1) (length s2)) (mapc (lambda (el) (member el s2)) s1)) ))
(24)

I. LISP Programming. You cannot use PROGs or any LOOP/DO constructs.

  a. Write a LISP function to take the dot product of two vectors each stored as LISP lists. The function returns the LISP atom ERROR! if the vectors do not have the same number of elements.

  Lisp→ (DOT_PRODUCT '(1 2 3 4) '(1 2 3 4))
  30
  Lisp→ (DOT_PRODUCT '(3 4 5) '(5 4 3))
  46
  Lisp→ (DOT_PRODUCT '(1 1) '(1))
  ERROR!

  (defun dot_product (v1 v2) (cond
                          ((null v1) nil)
                          ((null v2) nil)
                          ((= (length v1) (length v2)) (multv v1 v2))
                          (t 'ERROR!)))

(bdefun multv (v1 v2) (cond
                      ((null v1) 0)
                      (t (+ (* (first v1) (first v2)) (multv (rest v1) (rest v2)))))

  b. Write a LISP function which uses the DOT_PRODUCT function from part a above to multiply the \(n \times m\) matrix \(A\) by the \(m \times k\) matrix \(B\), where an \(n \times m\) matrix is represented internally as \(n\) lists of \(m\) characters each. Assume you have available the function TRANSPOSE which transposes a matrix represented this way. (Hint: You may wish to consider using MAPping function(s)). For example,

  Lisp→ (TRANSPOSE '( (1 2) (3 4) (5 6) ))
              ( (1 3 5) (2 4 6) )
  Lisp→ (MATRIX_PRODUCT '((1 2 3) (4 5 6)) '((1 1) (1 -1) (-1 1)))
              ( (0 2) (3 5) )
  Lisp→ (MATRIX_PRODUCT '((1 1) (1 -1) (-1 1)) '((1 2 3) (4 5 6)))
              ( (5 7 9) (-3 -3 -3) (3 3 3) )

  (defun matrix_product (A B) (mp A (transpose B)))
  (defun mp (m1 m2) (cond
                     ((null m1) nil)
                     (t (cons (mapcar #'(lambda (v) (dot_product (first m1) v)) m2) (mp (rest m1) m2)))))

  c. For extra credit, write the function TRANSPOSE from part b above.

  (defun transp (mat) (if (null (first mat)) nil (cons (mapcar #'first mat) (transp (mapcar #'rest mat))))))
(24)

I. LISP Programming. You cannot use PROGs or any LOOP/DO constructs.

(10) a. Write the function MAPPENDCAR, which is like MAPCAR, except that it uses APPEND instead of CONS to construct the list of results.

Lisp→> (DEFUN INCR(X) (LIST (+ X 1)))
Lisp→> (MAPPENDCAR #'INCR '(1 2 3))
Lisp→> (2 3 4)

(DEFUN MAPPENDCAR (F LIS) (COND
    ((NULL LIS) NIL)
    (T (APPEND (FUNCALL F (CAR LIS)) (MAPPENDCAR F (CDR LIS))))))

(4) b. Give the resulting symbolic expression(s) for the following LISP function evaluations. Show work for partial credit.

Lisp→ (MAPCAR #'INCR '(4 5 6)) RESULT: ___((5) (6) (7))___________
Lisp→ (MAPPENDCAR #'INCR '(4 5 6)) RESULT: ___(5 6 7)___________

(10) c. Using the function(s) MAPCAR or MAPPENDCAR give a definition for SETDIFF which when given two sets as input, returns all elements in the first set which are not in the second set.

Lisp→ (SETDIFF '(A B C D) '(B A))
     (C D)
Lisp→ (SETDIFF '(A B C D) '(D E F))
     (A B C)
Lisp→ (SETDIFF '(A B C) '(D E F))
     (A B C)

(DEFUN SETDIFF (S1 S2) (MAPPENDCAR (LAMBDA (EL) (IF (MEMBER EL S2) NIL (LIST EL))) S1))
I. LISP Programming. You cannot use PROGs or any LOOP/DO constructs.

a. Write the function `SUBST-SPLICE`, which is like `SUBST`, except that it “splices in” its first argument for the second.

```lisp
(defun subst (in out lis) (cond
    ((atom lis) lis)
    ((equal out (car lis)) (cons in (subst in out (cdr lis))))
    (t (cons (subst in out (car lis))
             (subst in out (cdr lis))))))
```

```lisp
(defun subst-spli-se (in out lis) (cond
    ((atom lis) lis)
    ((equal out (car lis)) (if (atom in) (cons in (subst-spli-se in out (cdr lis))))
                              (append in (subst-spli-se in out (cdr lis))))
    (t (cons (subst-spli-se in out (car lis))
             (subst-spli-se in out (cdr lis))))))
```

b. Give the resulting symbolic expression(s) for the following LISP function evaluations. Show work for partial credit.

```lisp
Lisp→ (FUNCALL #'(LAMBDA (N (** 2 N)) '(4 6 8))
         RESULT: ___(16 64 256)_______________________________
```

```lisp
Lisp→ (MAPCAR #'(LAMBDA (SX) (LIST (QUOTE →) SX)) '(GATORS ARE NUMBER 1))
       RESULT: ___( (→ GATORS) (→ ARE) (→ NUMBER) (→ 1)) _______________________________
```

c. Using the built-in function `MAPCAN`, which is like `MAPCAR` except it uses `APPEND` instead of `CONS`, give a definition for `INTERSECT` which when given two sets as input, returns all elements in the first set which are also in the second set, in order.

```lisp
(defun mapcan (f lis) (cond
    ((null lis) nil)
    (t (append (funcall f (car lis)) (mymapcar f (cdr lis))))))
```

```lisp
(defun intersect (s1 s2) (mapcan (lambda (el) (if (member el s2) (list el) nil)) s1))
```
LISP Programming. You cannot use PROGs or any LOOP/DO constructs.

1. Given the following association lists of adjective and noun opposites write a function FLIP to return the opposites of a simple two atom list.

Lisp→ ( SETF *ADJ-OPP* '((BIG LITTLE) (LITTLE BIG) (GOD BAD) (BAD GOOD) (HOT COOL) (COOL HOT)) )
((BIG LITTLE) (LITTLE BIG) (GOD BAD) (BAD GOOD) (HOT COOL) (COOL HOT))
Lisp→ ( SETF *NOUN-OPP* '((BROTHER SISTER) (SISTER BROTHER) (DOG CAT) (CAT DOG) (APPLE ORANGE) (ORANGE APPLE)) )
((BROTHER SISTER) (SISTER BROTHER) (DOG CAT) (CAT DOG) (APPLE ORANGE) (ORANGE APPLE))
Lisp→ ( FLIP 'BIG SISTER )
(LITTLE BROTHER)
Lisp→ ( FLIP 'BAD APPLE )
(GOOD ORANGE)
Lisp→ ( FLIP 'HOT DOG )
(COOL CAT)

(DEFUN FLIP (PHRASE) ( APPEND
 (CDR (ASSOC (CAR PHRASE) *ADJ-OPP*))
 (CDR (ASSOC (CAR (CDR PHRASE)) *NOUN-OPP*)) ))

2. Write a function that evaluates expressions written in infix notation, containing only Lisp functions that can take two arguments. Assume the expressions contain only numbers and defined XLISP functions. In this version dispense with type error checking completely. Example expressions and the values they should yield are:

Lisp→ (INFIX-EVAL ' (3 + 4) )
7
Lisp→ (INFIX-EVAL ' (3 * 4) )
12
Lisp→ (INFIX-EVAL ' (3 + (4 * 5)) )
23
Lisp→ (INFIX-EVAL ' ( (4 * 5) – (3 + (2 * 4)) ) )
9

(DEFUN INFIX-EVAL (EXP) ( IF (ATOM EXP)
 ;; ASSUME IT WAS A NUMBER IF IT IS AND ATOM EXP
 ;; FOR A LIST ASSUME IT IS (<EXP> <OP> <EXP> ).
 (FUNCALL (CADR EXP) )
 (INFIX-EVAL (CAR EXP))
 (INFIX-EVAL (CADDR EXP)) ) )

3. Using the built-in function MAPCAN, which is like MAPCAR except it uses APPEND instead of CONS, give a definition for a function to filter one or more lists, picking out only entries that satisfy some test (e.g., numbers). For extra credit (5) pass the test function as an argument to the filter function.

Lisp→ (FILTER ' ( A 3 B 2 4 C 7) )
(3 2 4 7)

(DEFUN FILTER(LIS) (MAPCAN #'(LAMBDA (ELEMENT)
 (IF (NUMBERP ELEMENT) (LIST ELEMENT) NIL)) LIS ))
I. Recursive LISP Programming. You cannot use PROGs or any LOOP/DO constructs.

Matrices and vectors are conveniently represented using arrays. When they get large, this can become unwieldy. In the case of sparse matrices/vectors storage is wasted on the large number of 0 entries, which also lead to useless arithmetic operations. Adding 0 and multiplying by 0 can be avoided if just the non-zero elements are stored. A sparse vector can be represented conveniently as a list of two-element sub-lists. Each sub-list contains an index and the corresponding component. The vector \[1.2,0,3.4,0,0,-6.7,0\]^T can be represented by the list \(((1 1.2) (3 3.4) (6 -6.7))\).

a. Develop a procedure to multiply a sparse vector by a scalar.
Lisp→ (SCALAR-BY-SPARSE-V 2 \'((1 1.2) (3 3.4) (6 -6.7)))

Lisp→ (((1 2.4) (3 6.8) (6 –13.4))

(DEFUN SCALAR-BY-SPARSE-V (SCALAR V)
  (IF (ZEROP SCALAR) ;SPECIAL CASE
    NIL
    (IF (ENDP V) ;TERMINATION?
      NIL
      (LET ((FV (FIRST V)))
        ;GRAB FIRST ENTRY.
        (CONS (LIST (FIRST FV) (* SCALAR (SECOND FV)))
          ;COPY INDEX & MULTIPLY.
          (SCALAR-BY-SPARSE-V SCALAR (REST V)) ) ) ) )
  ) ))
  ) )
  )

b. Develop a procedure to calculate the dot product of two sparse vectors.
Lisp→ (SPARSE-DOT-PRODUCT \'((1 2) (3 3) (6 4)) \'((1 1) (6 3)))

14

Lisp→ (((1 3) (3 3) (6 7))

(DEFUN SPARSE-DOT-PRODUCT (A B)
  (IF (OR (ENDP A) (ENDP B)) 0 ;TERMINATION
    (LET ((FA (FIRST A)) (FB (FIRST B)))
      ;GRAB COMPONENTS
      (COND (< (FIRST FA) (FIRST FB)) (SPARSE-DOT-PRODUCT (REST A) B))
      ( (< (FIRST FB) (FIRST FA)) (SPARSE-DOT-PRODUCT A (REST B)))
      (T (+ (* (SECOND FA) (SECOND FB)) (SPARSE-DOT-PRODUCT (REST A) (REST B)) ))))) ;RECURSE.

(DEFUN SPARSE-V-PLUS (A B) (COND
  ((ENDP A) B) ;TERMINATION
  ((ENDP B) A) ;TERMINATION
  (T (LET ((FA (FIRST A)) (FB (FIRST B)))
    ;GRAB COMPONENTS
    (COND (< (FIRST FA) (FIRST FB)) (CONS FA (SPARSE-V-PLUS (REST A) B))
      ( (< (FIRST FB) (FIRST FA)) (CONS FB (SPARSE-V-PLUS A (REST B)))
      (T (CONS (LIST (FIRST FA) (+ (SECOND FA) (SECOND FB)))
        ;ADD
        (SPARSE-V-PLUS (REST A) (REST B)) )))))) ;RECURSE.)
I. Recursive LISP Programming. You cannot use PROGs or any LOOP/DO constructs.

a. The covariance of two lists, $X$ and $Y$, of the same length, is defined as

$$(\text{covariance}) = \frac{n-1}{n} \times \left\{ \frac{\sum X_i Y_i}{n} - \left( \frac{\sum X_i}{n} \right) \left( \frac{\sum Y_i}{n} \right) \right\}$$

where

- $X_i$ is the $i$th element of list $X$,
- $Y_i$ is the $i$th element of list $Y$,
- $n$ is the number of items.

Write a function to compute the covariance of $X$ and $Y$. Hint: You can use `mapcar` to get the crossproduct of $X$ and $Y$, which is a list of $X_i Y_i$.

Lisp→ `(COVARIANCE '(1 2 3) '(4 5 6))
0.33333333333333304

(DEFUN COVARIANCE (LIS1 LIS2)
  (/ (- (MEAN (MAPCAR #'* LIS1 LIS2)))
      (* (MEAN LIS1) (MEAN LIS2)))
  (+ -1 (LENGTH LIS1)))

(DEFUN MEAN (LIS &OPTIONAL (SUM 0) (N 0))
  (COND ((NULL LIS) (IF (ZEROP N) 0 (/ SUM N)))
        (T (MEAN (CDR LIS) (+ SUM (CAR LIS)) (+ N 1)))))

b. Write a function called `list-props`, with one parameter that holds a list of names such as (mary bill ted patty). Each name in the list has two properties associated with it — age and sex. For each name in the list, this function creates a list consisting of the name and the value of the age and sex property for the name — for example, (mary 20 f) and (bill 19 m). The function returns a list of these lists (in the same order that the names appeared in the parameter list).

Lisp→ `(list-props '(mary bill))
((MARY 20 F) (BILL 19 M))
> (list-props 'bill mary))
((BILL 19 M) (MARY 20 F))

(DEFUN LIST-PROPS (LIS)
  (MAPCAR #'(LAMBDA (ITEM)
              (LIST ITEM (GET ITEM 'AGE) (GET ITEM 'SEX)))
           LIS))

c. Develop a function called `add-to-lis` that has two parameters. The first parameter holds a number and the second parameter holds a list of numbers. This function returns a list of sums that is created by adding the value of the first parameter to each number in the second parameter.

Lisp→ `(ADD-TO-LIS 45 '(2 98 –38))
(47 143 7)

(DEFUN ADD-TO-LIS (NUM NUMLIS) (MAPCAR #'(LAMBDA (NUM2) (+ NUM2 NUM)) NUMLIS))
I. Recursive LISP Programming. You cannot use PROGs or any LOOP/DO constructs unless specified.

a. Define a function `subset` that takes two arguments, a function and a list. `subset` should apply the function to each element of the list. It returns a list of all the elements of this list for which the function application returns non-nil. For example, {Hint: You may wish to consider the use of mapping & helper function(s)}

```
Lisp→ > (SUBSET 'NUMBERP '(A B 2 C D 3 E F))
(2 3)
```

```
(DEFUN MASK (M L) (MAPCAN #\(LAMBDA (TVAL EL) (IF TVAL (LIST EL))\) M L))
(DEFUN SUBSET (FUNC LIS) (MASK (MAPCAR FUNC LIS) LIS))
```

b. Write a function called `sub-splice`, like `subst`, but which “splices in” its first argument for the second. For example, {Hint: You do not need mapping & helper function(s)}

```
Lisp→ > (sub-splice '(1 2) 'b '(a b c))
(A 1 2 C)
Lisp→ > (sub-splice '(1 2) 'b '(a (b c) d))
(A (1 2 C) D)
```

```
(DEFUN SUB-SPlice (X Y Z) (COND
  ((ATOM Z) Z)
  ((EQUAL Y (CAR Z)) (APPEND X (SUB-SPlice X Y (CDR Z))))
  (T (CONS (SUB-SPlice X Y (CAR Z)) (SUB-SPlice X Y (CDR Z)))))
```

c. Use property lists to represent information about the cost and model numbers of a set of different makes of automobiles. For example, you might include that a Mercedes Benz 380SL costs $50,000, a BMW 320i costs $35,000, a Honda Accord costs $18,000 and a Honda Civic costs $12,000. Now write a function that, given a list of cars as input, returns the model of the least expensive car. {You can use any LISP construct you want, such as Prog, DO, Lambda, mapping fcns, etc.}

```
(PUTPROP 'CAR1 'MB 'MAKE) (PUTPROP 'CAR1 '380SL 'MODEL) (PUTPROP 'CAR1 50000 'COST)
(PUTPROP 'CAR2 'BMW 'MAKE) (PUTPROP 'CAR2 '320i 'MODEL) (PUTPROP 'CAR2 35000 'COST)
(PUTPROP 'CAR3 'HONDA 'MAKE) (PUTPROP 'CAR3 'Accord 'MODEL) (PUTPROP 'CAR3 18000 'COST)
(PUTPROP 'CAR4 'HONDA 'MAKE) (PUTPROP 'CAR4 'Civic 'MODEL) (PUTPROP 'CAR4 12000 'COST)
(SETQ CAR-LIST '(CAR1 CAR2 CAR3 CAR4))
LISP→ > (LEAST-EXPENSIVE CAR-LIST)
(HONDA Civic 12000)
```

```
(DEFUN LEAST-EXPENSIVE (CARLIST) (DO ((CL CARLIST (CDR CL)) (RESULT (CAR CARLIST)))
  ((NULL CL) (MAPCAR #\(LAMBDA (EL) (GET RESULT EL)) '(MAKE MODEL COST)))
  (IF (> (GET RESULT 'COST) (GET (CAR CL) 'COST)) (SETQ RESULT (CA R CL)))))
```
(33) Recursive LISP Programming. You cannot use PROGs or any LOOP/DO constructs unless specified.

(12) a. Write the function EQUAL-SYMBOLS of two arguments that says any two numbers are equal, a symbol is only equal to itself, and two lists are equal if all their elements are recursively EQUAL-SYMBOLS. For example:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(EQUAL-SYMBOLS 2.2 3)</code></td>
<td>T</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS 'B 4 (C D (2.2)) E) </code>(B 2 (C D (0)) E))`</td>
<td>T</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS 2.2 'A)</code></td>
<td>NIL</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS 'B 4 (C D (2.2)) E) </code>(B 4 (3 D (2.2))) E))`</td>
<td>T</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS 'A) 'A)</code></td>
<td>NIL</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS NIL 'A)</code></td>
<td>NIL</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS 'A 'A)</code></td>
<td>NIL</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS 'A (CAR '(A B)))</code></td>
<td>NIL</td>
</tr>
<tr>
<td><code>(EQUAL-SYMBOLS NIL (CAR '(A B)))</code></td>
<td>T</td>
</tr>
</tbody>
</table>

```lisp
defun equal-symbols (s ex1 ex2)
  (cond
    ((numberp ex1) (numberp ex2))
    ((atom ex1) (eq ex1 ex2))
    ((atom ex2) nil)
    ((null ex1) (null ex2))
    ((null ex2) nil)
    ((equal-symbols (car ex1) (car ex2))
      (equal-symbols (cdr ex1) (cdr ex2)))
    )
```

(10) b. Give a LISP definition, using a mapping function to calculate the union between two sets S1 and S2. The union of two sets is defined as the set in which every element of the union is a member of either the 1st or the 2nd set.

```lisp
(defun union (s1 s2)
  (append (mapcan #'(lambda (el) (if (member el s2) nil (list el))) s1) s2))
```

(10) c. Write a function, MAKESET, which takes a simple list as input and makes a set using an accumulator. Make sure your answers are in the correct order.

```lisp
(defun make-set (lis &optional ans)
  (cond
    (null lis) ans
    ((member (car lis) (cdr lis)) (make-set (cdr lis) ans))
    (else (make-set (cdr lis) (append ans (list (car lis))))))
```