Prof. Dr. H. Stoyan
Universität Erlangen-Nürnberg
Institut für Mathematische Maschinen
und Datenverarbeitung (Informatik VIII)
Am Weichselgarten 9
91058 Erlangen

QUANTUM THEORY PROJECT
FOR RESEARCH IN ATOMIC, MOLECULAR AND SOLID STATE
CHEMISTRY AND PHYSICS
UNIVERSITY OF FLORIDA, GAINESVILLE, FLORIDA

AN INTRODUCTION TO LISP

by

Arnold K. Griffith

PREPRINT NO. 31
November 15, 1962

This work was supported in part
by a grant from the
NATIONAL SCIENCE FOUNDATION

INTRODUCTION

This volume is intended to be an introduction to LISP programming. In addition, the last section constitutes a revision of the HANDBOOK OF LISP FUNCTIONS (RIAS technical report 61-11).

This volume was prepared during an 8-week summer programming institute held by Mr. H. V. McIntosh at the University of Florida, summer 1962.

Gratitude is due the University for the use of their 709 to run the examples of the last section; and to the writers of the aforementioned MANUAL and Mr. McIntosh from whom all function definitions, certain function descriptions, and other ideas were lifted verbatim.

Gainesville,
August, 1962

Arnold K. Griffith

CONTENTS

LISP

The notation of LISP consists entirely of lists. A list is a series of elements which is enclosed in parentheses:

(list element list element .....)

List elements may be lists or atomic symbols. An atomic symbol is a string of characters unbroken by spaces or parentheses:

POTRZEBIE

54321BANG

(etc.)

Note that a pair of adjacent atoms on a list must be separated or they will be taken as a single atom. This is done with a space. The parentheses which must surround lists will distinguish list elements in other cases. List example:

((APE CALL) C (D (E F)))

A simple diagram of the type shown in figure 1. may serve to schematize a list. Each dot, except for the top one which stands for the whole list, stands for a list element, whose form in list notation is written by the dot.

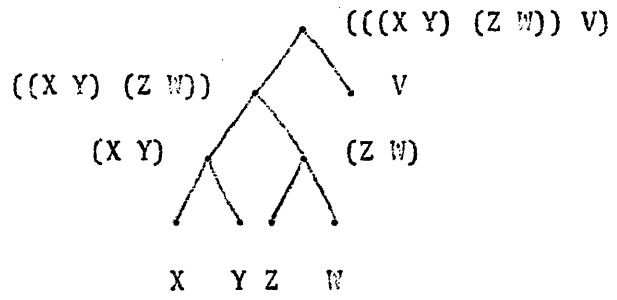
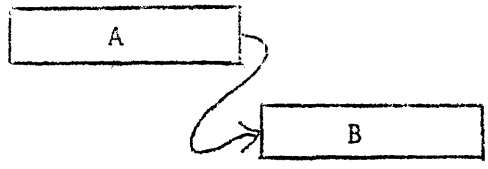
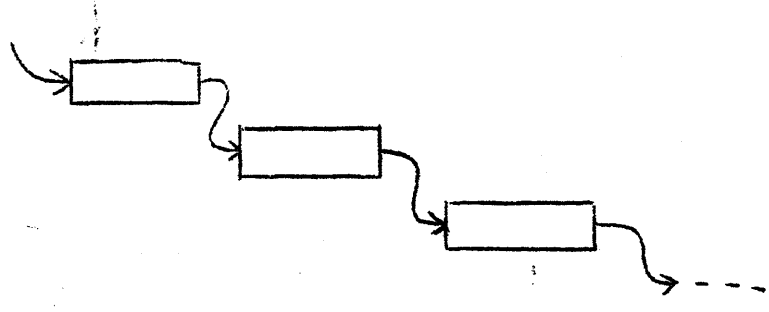


FIG. 1.

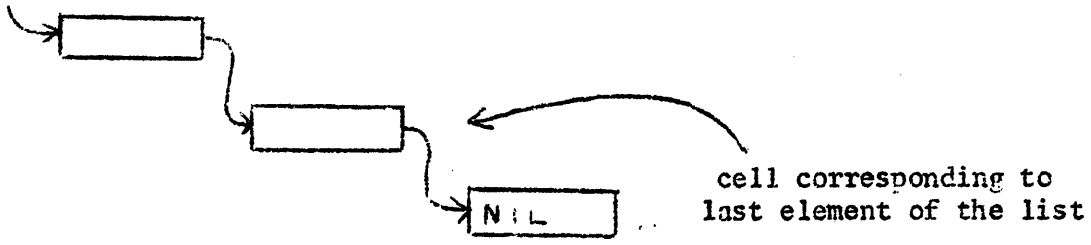
A diagrammatic representation of a list which gives some insight into how lists are stored in the machine uses a system of rectangles and arrows. Information is stored in many machines in a series of thousands or tens of thousands of identical cells, each accessible to the processing section of the machine by a unique address. A cell can hold two addresses; and most cells used by the LISP system are divided in a particular manner into two halves, in each of which an address is stored. These halves are differentiated by calling one the left half and the other the right half. The use of the terms right and left comes from the system of notation about to be described. A cell is represented by a rectangle; and the situation of a right or left contents of a cell being the address of another cell is represented by an arrow from the appropriate side of the first cell to the second cell. The following illustrates a cell, A, whose right contents is the address of the cell B:



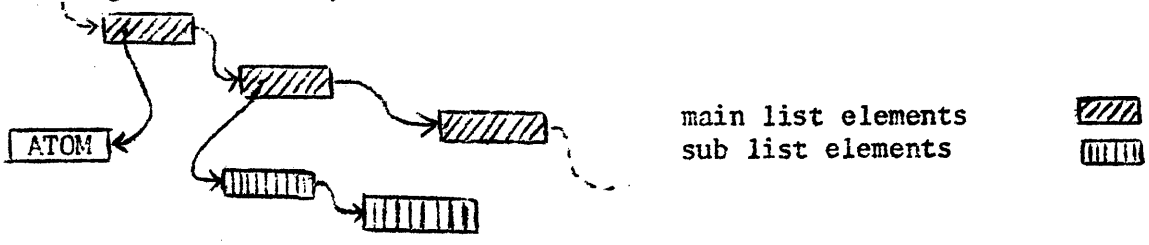
A list is represented by a series of cells, one for each element of the list. For each list element the contents of the right half of its corresponding cell is the address of the cell corresponding to the next element of the list:



An exception is made in the case of the last element, where the right address is that of a particular cell chosen to indicate the end of a list:



The contents of the left half of each cell is an address which indicates where the cell's corresponding element is stored: if the element is an atom, the address may be considered to be the address of a cell containing that atom; if the element is another list, the address is the address of the cell corresponding to the first element on that list. The relationships are again denoted by arrows:



A function is a list whose first element is an atom, the name of the function; and which is equal, in a sense which will be made clearer later, to some other list or atom. This second list is determined by the definition of the function, by the arguments (the remaining elements of the list), and sometimes by context.

Following is a description of the five so-called primitive functions:

1. (ATOM X) is equal to a list containing one element, either T or F. It takes the value T (true) if X is an atomic symbol, otherwise F:

(ATOM TELEMACHUS) = T

(ATOM (ZOD)) = F

(ATOM (IBM BAD)) = F

The contents of an empty list is regarded as an atom. This is an example of a predicate, a function which takes the value T or F.

2. (EQ X Y) is another predicate. It is equal to T if and only if both arguments (X and Y) are equal atoms:

(EQ ZUP ZUP) = T

(EQ SNORT ZEUS) = F

(EQ (MAD) MAD) = F

It gives weird results if neither argument is an atom.

3. (CAR X) is equal to the first element of the list X. It may be an atom or a list:

(CAR (AB ZUT)) = AB

(CAR ((A B) ZUT)) = (A B)

CAR of an atom is undefined.

4. (CDR X) is equal to a list of all the elements of X except the first:

(CDR (DON'T GO AWAY)) = (GO AWAY)

(CDR (BAH)) = ( )

CDR of an atom is undefined.

5. (CONS A B) is equal to a list whose first element is A, and whose subsequent elements are the elements of B:

(CONS (POT) (R Z E B I E)) =

((POT) R Z E B I E)

(CONS A B), where B is an atom, is undefined.

It will be found, upon writing programs in LISP, that composites of CAR and CDR are necessary and can get very lengthy. These can be abbreviated in the following manner: (CAR (CDR L)) can be written as

(CADR L), and similarly, (CAR (CAR (CAR (CDR L)))) becomes (CAADR L). Such abbreviations with up to four letters between the C and the R are defined and used in the same way as the primitive functions.

Functions which will do other tasks that the primitive functions must be written in terms of the primitive functions, with the aid of the programming functions, in the manner which will now be described. The programming functions are: DEFINE, LAMBDA, COND, AND, OR, NOT, NULL, QUOTE, and LIST.

To define a function or series of functions, we write a list whose first element is the atom DEFINE:

(DEFINE .....)

Its subsequent elements are lists of two elements, the first of which is an atom, the name you wish to give to the function; and the second of which is that function's definition:

```
(DEFINE
  (name definition)
  (name definition)
      .......
  (name definition))
```

A function's definition is in turn a list of three elements: the atom LAMBDA, an ordered list of the function's arguments, and the actual specification:

(LAMBDA (args.) (specification))

The specification is generally* a list whose first element is the atom COND, and whose subsequent elements are lists of two elements: a predicate, and an expression:

---

* If the definition is not conditional, instead of (COND (pred. exp.) (pred. exp.) - - - _ simply an expression appears. cf.

```
(COND (predicate expression)
      (predicate expression)
          .........
      (predicate expression)
```

The function takes the value of the expression following the first true predicate. The total structure of a DEFINE function is, then:

```
(DEFINE
 (FUNCTA (LAMBDA (args.) (COND
  (predicate expression)
  (predicate expression)
     ..............
  (predicate expression))))
  ........................
 (FUNCTA (LAMBDA (args.) (COND
  (predicate expression)
  (predicate expression)
     ..............
  (predicate expression)))))
```

The predicates and expressions are composites of primitive functions; of the predicates AND, OR, NOT and NULL; and of the functions QUOTE and LIST.

(AND predicate$_1$ predicate$_2$ predicate$_3$ .....) is a predicate which is equal to true if and only if all its arguments are true predicates. It examines its arguments sequentially and becomes equal to F upon finding an argument which is not a true predicate, if it does.

(OR predicate$_1$ predicate$_2$ predicate$_3$ .....) is a predicate which is equal to true if and only if any of its arguments is a true predicate. It examines its arguments sequentially in the same way that AND does.

(NOT predicate) is equal to T if its argument is a false predicate, and F if its argument is a true one.

(NULL L) is equal to T if L is a null list, otherwise false.

(QUOTE atomic symbol) is equal to its atomic symbol argument, if the argument is an atomic symbol.

(LIST arg.$_1$ arg.$_2$ arg.$_3$ .....) is equal to a list of its arguments.

As was said before, a function is equal (in a sense which has been partially illustrated) to some list or atom. The function DEFINE may be thought of as equal to one of the functions it defines, the particular one, and its particular arguments being determined by context. The context is the rest of the LISP program in which the DEFINE expression appears.

The totality of a LISP program consists of a DEFINE expression, just described, an APPLY expression, which will shortly be explained, and possibly TRACLIS or UNTRACLIS expressions, which will be defined later. The first two types of expressions might correspond to:

$$f(x) = g(x) + 3m(x) + 4$$

$$g(x) = x^2$$

$$m(x) = 3g(x)$$

and

$$f(5)=254$$

respectively.

An APPLY statement is a list of three elements: the atom-name APPLY; the atom-name of the function which the program is interested in evaluating; and a list of the arguments for that function. In the context of the program, the DEFINE expression becomes equal to the function specified by the APPLY, and the APPLY expression becomes the value of that function for the given arguments.

An example of what has so far been presented, let us write the following program:

```
(DEFINE
 (TRANSLATE (LAMBDA (A B) (COND
  ((NULL A) (LIST)
  ((T) (CONS (LOOKUP (CAR A) B)
             (TRANSLATE (CDR A) B))))))
 (LOOKUP (LAMBDA (A B) (COND
  ((EQ A (CAAR B)) (CDAR B))
  ((T) (LOOKUP A (CDR B)))))))

(APPLY TRANSLATE ((THE DOG ATE THE STEAK)
   ((STEAK VIFTEK) (DOG CHIEN) (ATE MANGEAIT) (THE LE))))
```

The result of this program will be TRANSLATE of the two arguments given. To evaluate TRANSLATE we first test the first predicate of its definition. Since the first argument is not a null list, the predicate is false, and we try the next one, T in this case is a pseudo-predicate, which is alsays true. (it may be read as "otherwise".) In this case, since no prior predicate of the definition was true, it causes the definition paired with it to be the definition of TRANSLATE, and ultimately of APPLY. TRANSLATE is then equal to LOOKUP of the first element of the first argument CONSed with TRANSLATE of the remainder of the first argument. This second TRANSLATE is similarly evaluated. When we get to the point when we must take TRANSLATE of a list of only one atom, we again CONS LOOKUP of that atom with translate of CDR of the list, which is the empty list. To evaluate this TRANSLATE, we again go down the list of predicates, but this time (NULL A) is true. Thus TRANSLATE id defined equal to the list, which needs no further evaluation. TRANSLATE is then equal to a list in one to one correspondence with the first argument. Each element of this list is equal to LOOKUP of the corresponding element on the first argument. LOOKUP of the atoms of the first argument must now be evaluated. Given an atomic symbol from the first argument, it looks to see if it is equal to the first element of the first element of

the second argument. If so, LOOKUP in that case is equal to the second atom in that element, which would be its correspond in the other language. If the first predicate is not true, LOOKUP is equal to LOOKUP of the atom and of CDR of the second argument. This process must eventually come upon the right pair in the second argument, assuming it exists and make LOOKUP equal to the other-language correspond of the given atom-word. Thus, TRANSLATE gives a word for word translation of a list of atoms.

In conclusion, let us look at some slightly more specialized pre-defined functions. The distinction between pre-defined functions and functions defined by a define statement may here be more clearly described, and should be understood: pre-defined functions may appear in a program without definition in the DEFINE expression: they are recognized and appropriately dealt with by whatever operates on the LISP program; all other functions used in a program must be defined in the DEFINE expression of that program.

TRACLIS aids in debugging programs. To use the function TRACLIS, one inserts a TRACLIS expression in the program along with DEFINE and APPLY expressions. It constist of a list whose first element in the atom TRACLIS, and whose subsequent elements are names of functions to be TRACLISed:

```
(TRACLIS FUNCTA FUNCTB)

(APPLY FUNCTA)

(DEFINE
 (FUNCTA.....)
 (FUNCTB.....
     .....

 (FUNCTZ.....))
```

TRACLIS applied to a function will list the successive values the function

takes as it is being evaluated.

UNTRACLIS may be used if there are several APPLYs to keep from TRACLISing all of them. All APPLYs after a TRACLIS but before the following UNTRACLIS will be TRACLISed, others will not.

DISINTEGRATE is a function of an atom, and is equal to a list of symbols in that atom:

(DISINTEGRATE SOUTH)=(S O U T H)

REINTEGRATE is similarly defined for the opposite purpose:

(REINTEGRATE (D O G)) = DOG

FUNCTION and LAMBDA must be used when a function appears as an argument of another function, as in the case of FORODD which is a function of a predicate and a list:

....(FORODD (FUNCTION (LAMBDA (X) (ATOM X))) L).....

instead of

....(FORODD ATOM L)

as might be expected.

LAMBDA is also used in function definitions to avoid the re-evaluation of certain expressions. A good example is the function EXTRACT:

```
(EXTRACT (LAMBDA (P L) (COND
 ((NULL L) (LIST (LIST) (LIST))
 ((P (CAR L)) ((LAMBDA (X) (CONS (CONS (CAR L) (CAR X))
  (CDR X))) (EXTRACT P (CDR L))))
 ((T) ((LAMBDA (X) (LIST (CAR X) (CONS (CAR L) (CADR X))))
  (EXTRACT P (CDR L)))))))
```

Here, ((LAMBDA (X) (CONS (CONS (CAR L) (CAR X)) (CDR X))) (EXTRACT P (CDR L))) is used in place of (CONS (CONS (CAR L) (CAR (EXTRACT P (CDR L)))) (CDR (EXTRACT P (CDR L)))) to avoid calculation (EXTRACT P (CDR L)) twice.

TWO THEORETICAL POINTS

At this point we may bring up a fundamental fact about LISP which has required the preceding discussion as illustration. Notice that in LISP a function may be defined in terms of itself. This is not possible in most languages, which are iterative (one instruction after another). This self-definition, or recursion, marks the fundamental difference between LISP and most other programming languages. Some experience with programming in LISP may be required before the reader will fully appreciate the importance and elegance of recursion.

Another aspect which differentiates LISP from many other languages is the fact that a "universal" function may be written in the language. A universal function in a language is one which will take a program in that language as input, and give as output the value expected from that program. In LISP this amounts to a definition of APPLY written in LISP:

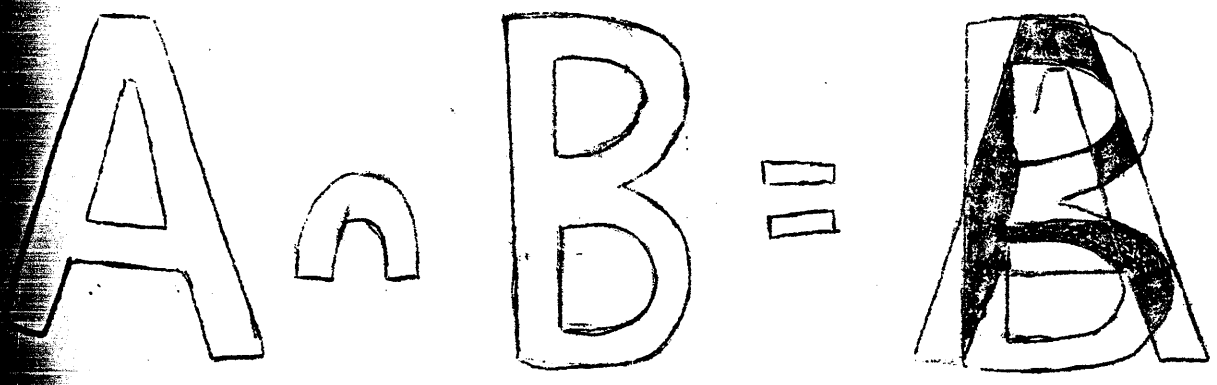
```
(APPLY (LAMBDA (E)
 (EVAL (CONS (CAR E) (APPQ (CDR E))) (ALIST*))))
(EVAL (LAMBDA (E ALIST) (COND
 ((NULL E) (LIST))
 ((ATOM E) (ASSOC E ALIST))
 ((NULL (CAR E)) (CADR E))
 ((ATOM (CAR E)) ((LAMBDA (X) (COND
  ((EQ X (QUOTE T)) (T))
  ((EQ X (QUOTE F)) (F))
  ((EQ X (QUOTE CAR)) (CAR (EVAL (CADR E) ALIST)))
  ((EQ X (QUOTE CDR)) (CDR (EVAL (CADR E) ALIST)))
  ((EQ X (QUOTE CONS )) (CONS (EVAL (CADR E) ALIST) (EVAL (CADDR E) ALIST)))
  ((EQ X (QUOTE ATOM)) (ATOM (EVAL (CADR E) ALIST)))
  ((EQ X (QUOTE EQ)) (EQ (EVAL (CADR E) ALIST) (EVAL (CADDR E) ALIST)))
  ((EQ X (QUOTE QUOTE)) (CADR E))
  ((EQ X (QUOTE LIST)) (EVLIS (CDR E) ALIST))
  ((EQ X (QUOTE COND)) (EVCON (CDR E) ALIST))
  ((EQ X (QUOTE NULL)) (NULL (EVAL (CADR E) ALIST)))
  ((EQ X (QUOTE FUNCTION)) (LIST (QUOTE FUNARG) (CADR E) ALIST))
  ((EQ X (QUOTE AND)) (EVAND (CDR E) ALIST))
  ((EQ X (QUOTE OR)) (EVOR (CDR E) ALIST))
```

```
((ATOM X) (EVAL (CONS (ASSOC X ALIST) (CDR E)) ALIST))
((EQ (CAAR E) (QUOTE LAMBDA)) (EVAL (CADDAR E) (EVLAPPEND (CADAR E) (CDR E) ALIST)))
((EQ (CAAR E) (QUOTE LABEL)) (EVAL (CONS (CADDAR E) (CDR E)) (CONS
                                        (CADR E) (CONS (CADDAR E) ALIST))))
((EQ (CAAR E) (QUOTE FUNARG)) (EVAL (CONS (CADAR E) (EVAPQ (CDR E)
                                        ALIST))) (CADDAR E)))
((T) (EVAL (CONS (EVAL (CAR E) ALIST) (CDR E)) ALIST)))))

(EVLIS (LAMBDA (L ALIST) (COND
 ((NULL L) (LIST))
      ((T) (CONS (EVAL (CAR L) ALIST) (EVLIS (CDR L) ALIST))))))

(EVCON (LAMBDA (L ALIST) (COND
  ((NULL L) (PRINT (QUOTE (UNSATISFIED CONDITIONAL))))
  ((EVAL (CAAR L) ALIST (EVAL (CADAR L) ALIST))
  ((T) (EVCON (CDR L) ALIST)))))

(EVAND (LAMBDA (L ALIST)
  (OR (NULL L) (AND (EVAL (CAR L) ALIST) (EVAND (CDR L) ALIST)))))

(EVOR (LAMBDA (L ALIST)
 (AND (NOT (NULL L)) (OR (EVAL (CAR L) ALIST) (EVOR (CDR L) ALIST)))))

(EVALAPPEND (LAMBDA (U V L) (COND
 ((AND (NULL U) (NULL V)) L)
 ((NULL U) (LIST (QUOTE (EXCESS ARGUMENTS)) V))
 ((NULL V) (LIST (QUOTE (EXCESS VARIABLES)) U))
 ((T) (CONS (CAR U) (CONS (EVAL (CAR V) ALIST) (EVALAPPEND (CDR U) (CDR V) L)))))))

(ASSOC (LAMBDA (X L) (COND
 ((NULL L) (LIST (QUOTE (UNDEFINED ATOM)) X (QUOTE ALIST) ALIST))
 ((EQ X (CAR L)) (CADR L))
 ((T) (ASSOC X (CDDR L))))))

(APQ (LAMBDA (L) (COND
 (NULL L) (LIST))
 ((T) (CONS (LIST (QUOTE QUOTE) (CAR L)) (APQ (CDR L)))))))
```

SOME DEBUGGED FUNCTIONS

SET THEORY.....

SET THEORY.....



(ELEMENT X L)

I. ELEMENT takes the value T if the given element is an element of the given list; otherwise it takes the value F.

II. ELEMENT is a predicate of:

X, a list element,

L, a list.

III. definition:

```
(ELEMENT (LAMBDA (X L)
 (AND (NOT (NULL L))
  (OR (EQUAL X (CAR L)) (ELEMENT X (CDR L))))))
```

IV. ELEMENT uses only pre-defined functions

V. examples:

```
(APPLY ELEMENT ((E L) ((E L) E (M (E N)) T)))
T

(APPLY ELEMENT (L ((EL) E (M (E N)) T)))
F

(APPLY ELEMENT (() ((E L) E (M (E N)) T)))
F
```

(INTERSECTION L)

I. INTERSECTION makes a list of all elements common to all the given lists.

II. INTERSECTION is a function of:

L, a list of any number of lists.

III. definition:

```
(INTERSECTION (LAMBDA (L) (COND
 ((NULL L) (QUOTE UNIVERSAL))
 ((T) (POSSESSING (FUNCTION (LAMBDA (U) (FORALL (FUNCTION (LAMBDA (V)
                                  (ELEMENT U V))) L))) (CAR L))))))
```

IV. INTERSECTION uses:

POSSESSING, FORALL.

V. examples:

```
(APPLY INTERSECTION (((B L E S T) (E A T S))))
(E S T)

(APPLY INTERSECTION ((() ())))
()

(APPLY INTERSECTION (((IBM IBB UBB) (BAM BUM IBM))))
(IBM)

(APPLY INTERSECTION (()))
UNIVERSAL
```

(UNION L)

I. UNION makes a list of all the elements on the given lists.

II. UNION is a function of:

L, a list of any number of lists.

III. definition:

```
(UNION (LAMBDA (L) (COND
 ((NULL L) (LIST))
 ((NULL (CDR L)) (CAR L))
 ((T) (UNION (CONS (ACCUMULATE (CADR L) (CAR L)) (CDDR L)))))))
```

IV. UNION uses:

ACCUMULATE.

V. examples:

```
(APPLY UNION (((T A N) (U N R E S T))))
(S E R U T A N)

(APPLY UNION (((A B C D E) (C D E F G))))
(C F A B C D E)

(APPLY UNION (((A B C D E) ())))
(A B C D E)
```

(DELTA L)

I. DELTA makes a list of all the elements on an odd number of the given lists.

II. DELTA is a function of:

L, a list of any number of lists.

III. definitionL

```
(DELTA (LAMBDA (L)
 (POSSESSING (FUNCTION ( LAMBDA (U)
  (FORODD (FUNCTION (LAMBDA (V) ELEMENT U V))) L))) (UNION L))))
```

IV. DELTA uses:

POSSESSING, FORODD, and ELEMENT.

V. examples:

```
(APPLY DELTA (((Z E B R A) (R O D) (D O T) (D A B) (T O E))))
 (D O Z)
```

(SUBSET S L)

I. SUBSET takes the value T if all the elements of the first list are elements of the second, F otherwise.

II. SUBSET is a predicate of:

S, a list,

L, a list.

III. definition:

```
(SUBSET (LAMBDA (S L) (COND
 ((NULL S) (T))
 ((ELEMENT (CAR S L) (SUBSET (CDR S) L))
 ((T) (F)))))
```

IV. SUBSET uses:

ELEMENT.

V. examples:

```
(APPLY SUBSET ((A) (A B C D)))
T

(APPLY SUBSET (() (A N Y T H I N G)))
T
```

(CARTESIAN L)

I. CARTESIAN writes a list of all possible n-tuples such that the first element of an n-tuple is from the first list n of the given list, the second from the second, etc.

II. CARTESIAN is a function of:

L, a list of any number of lists.

III. definition:

```
(CARTESIAN (LAMBDA (L)
 ((NULL L) (LIST))
 ((NULL (CDR L)) (MAPCAR (QUOTE LIST) (CAR L)))
 ((T) ((LAMBDA (X) (CARTESIAN* (CAR L) X X)) (CARTESIAN (CDR L)))))))
```

IV. CARTESIAN uses:

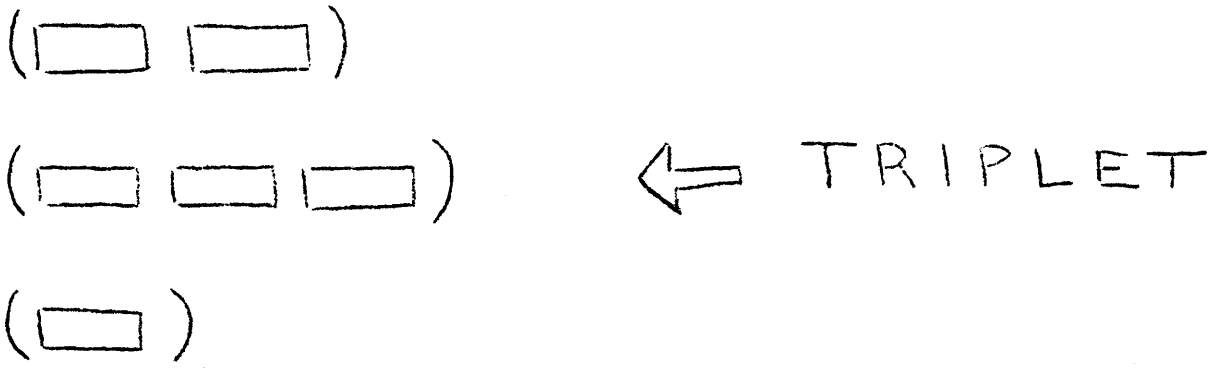
CARTESIAN* MAPCAR

V. examples:

```
(APPLY CARTESIAN (((I) (S T N T))))
((I S) (I T) (I N) ( I T))

(APPLY CARTESIAN (((M SS SS PP) (I))))
((M I) (SS I) (SS ()) (PP I))

(APPLY CARTESIAN (((1 2 3) (A B C) X Y Z))))
((1 A X) (1 A Y) (1 A Z) (1 B X) (1 B Y) (1 B Z) (1 C X)
(1 C Y) (1 C Y) (2 A X) ( 2 A Y) (2 A Z) (2 B X) (2 B Y) (2 B Z
) (2 C X) (2 C Y) ( 2 C Z) (3 A X) (3 A Y) (3 A Z) (3 B X)
(3 B Y) (3 B Z) (3 C X) (3 C Y) (3 C Z))
```

PREDICATES TO TEST A LIST

FOR A CONFIGURATIONAL PROPERTY.....



(ULTRASINGLET L)

I. ULTRASINGLET takes the value T if a given list has at least one element; F otherwise.

II. ULTRASINGLET is a predicate of:

L, a list.

III. definition:

```
(ULTRASINGLET (LAMBDA (L)
 (NOT (OR (ATOM L) (NULL L)))))
```

IV. ULTRASINGLET uses only pre-defined functions.

V. examples:

```
(APPLY ULTRASINGLET ((A B)))
T

(APPLY ULTRASINGLET ((A)))
T

(APPLY ULTRASINGLET (A))
F
```

(ULTRADOUBLET L)

I. ULTRADOUBLET takes the value T if the given list has at least two elements; F otherwise.

II. ULTRADOUBLET is a predicate of:

L, a list.

III. definition:

```
(ULTRADOUBLET (LAMBDA (L)
 (NOT (OR (ATOM L) (NULL L) (NULL (CDR L))))))
```

IV. ULTRADOUBLET uses only pre-defined functions.

V. examples:

```
(APPLY ULTRADOUBLET ((A)))
F

(APPLY ULTRADOUBLET (A))
F

(APPLY ULTRADOUBLET ((A B)))
T
```

(ULTRATRIPLET L)

I. ULTRATRIPLET takes the value T if the given list has three or more elements; F otherwise.

II. ULTRATRIPLET is a predicate of:

L, a list.

III. definition:

```
(ULTRATRIPLET (LAMBDA (L)
 (AND (ULTRADOUBLET L) (NOT (NULL (CDDR L))))))
```

IV. ULTRATRIPLET uses only pre-defined functions.

V. examples:

```
(APPLY ULTRATRIPLET ((A B C)))
T

(APPLY ULTRATRIPLET ((A B)))
F

(APPLY ULTRATRIPLET (((A B))))
F

(APPLY ULTRATRIPLET ((A)))
F
```

(SINGLET L)

I. SINGLET takes the value T if the given list has exactly one element;

F otherwise.

II. SINGLET is a predicate of:

L, a list.

III. definition:

```
(SINGLET (LAMBDA (L)
 (AND (ULTRASINGLET L) (NULL (CDR L)))))
```

IV. SINGLET USES:

ULTRASINGLET.

V. examples:

```
(APPLY SINGLET ((A)))
T

(APPLY SINGLET (A))
F

(APPLY LINGLET ((A B)))
F
```

(DOUBLET L)

I. DOUBLET takes the value T if the given list has exactly two elements; F otherwise.

II. DOUBLET is a predicate of:

L, a list.

III. definition:

```
(DOUBLET (LAMBDA (L)
 (AND (ULTRADOUBLET L) (NULL (CDDR L)))))
```

IV. DOUBLET uses:

ULTRADOUBLET.

V. examples:

```
(APPLY DOUBLET ((A B)))
T

(APPLY DOUBLET ((A B C)))
F
```

(TRIPLET L)

I. TRIPLET takes the value T if and only if the given list has exactly three elements; F otherwise.

II. TRIPLET is a predicate of:

L, a list.

III. definition:

```
(TRIPLET (LAMBDA (L)
 (AND (ULTRATRIPLET L) (NULL (CDDDR L)))))
```

IV. TRIPLET uses:

ULTRATRIPLET.

V. examples:

```
(APPLY TRIPLET ((A B C)))
T

(APPLY TRIPLET ((A B C D)))
F
```

(ODDPLET L)

I. ODDPLET takes the value T if the given list han an odd number of elements F otherwise.

II. ODDPLET is a predicate of:

L, a list.

III. definition:

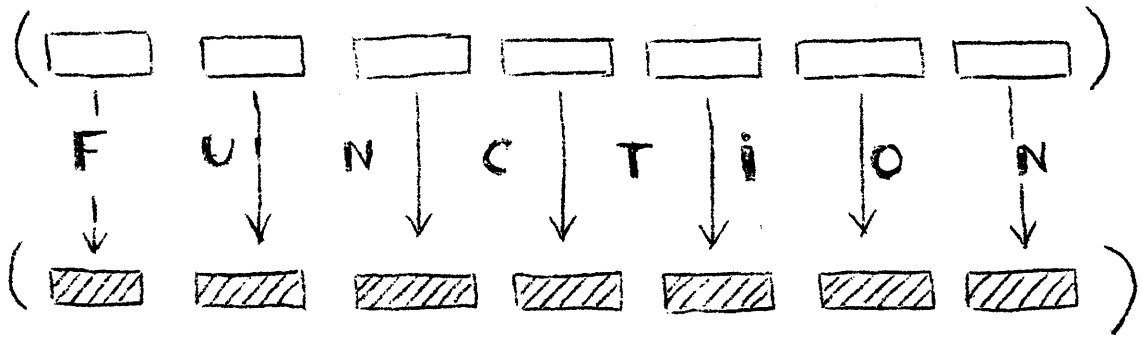
```
(ODDPLET (LAMBDA (L)
 (NOT (OR (NULL L) (ODDPLET (CDR L))))))
```

IV. ODPLET uses only pre-defined functions.

V. examples:

```
(APPLY ODDPLET ((A B C D)))
F

(APPLY ODDPLET ((A B C D E)))
T

(APPLY ODDPLET (((A B C D E))))
T
```

FUNCTIONS WHICH APPLY OTHER FUNCTIONS TO THE ELEMENTS OF A LIST.....



(MAPCAR G L)

I. MAPCAR applies a given function to every element of a given list and lists the results.

II. MAPCAR is a function of:

G, a function,

L, a list.

III. definition:

```
(MAPCAR (LAMBDA (G L) (COND
 ((NULL L) (LIST))
 ((T) (CONS (G (CAR L)) (MAPCAR G (CDR L)))))))
```

IV. MAPCAR uses only pre-defined functions.

V. examples:

```
(APPLY MAPCAR ((FUNCTION (LAMBDA (X) (CAR X))) ((I C) (B) (M))))
(I B M)

(APPLY MAPCAR ((FUNCTION (LAMBDA (X) (EQ X (QUOTE Y)))) (A B C Y D)))
(F F F T F)
```

(MAPCON G L)

I. MAPCON applies a function to a whole list, then to CDR of the list, and so on. The lists resulting are joined together.

II. MAPCON is a function of:

G, a function,

L, a list.

III. definition:

```
(MAPCON (LAMBDA (G L) (COND
 ((NULL L) (LIST))
 ((T) (APPEND (G L) (MAPCON G (CDR L)))))))
```

IV. MAPCON uses:

APPEND.

V. examples:

```
(APPLY MAPCON (CDR (A B C D)))
(B C D C D D)

(APPLY MAPCON ((FUNCTION (LAMBDA (X) (CDR X))) (A B C D)))
(B C D C D D)
```

(MAPLIST G L)

I. MAPLIST applies a function to a list, then to CDR of the list, and so on. The results of each application form the elements of a new list.

II. MAPLIST is a function of:

G, a function,

L, a list.

III. definition:

```
(MAPLIST (LAMBDA (G L) (COND
 ((NULL L) (LIST))
 ((T) (CONS (G L) (MAPLIST G (CDR L)))))))
```

IV. MAPLIST uses only pre-defined functions.

V. examples:

```
(APPLY MAPLIST ((FUNCTION (LAMBDA (X) (ATOM X))) (A B C D)))
(F F F F)

(APPLY MAPLIST ((FUNCTION (LAMBDA (X) (CAR X))) (A B C D)))
(A B C D)
```

(FOREACH G X L)

I. FOREACH applies a function of two arguments to the successive elements of a given list and simultaneously to a second list; and lists the results.

II. FOREACH is a function of:.

G, a function

X, a list,

L, a list.

III. definition:

```
(FOREACH (LAMBDA (G X L) (COND
 ((NULL X (LIST))
 ((T) (CONS (G (CAR X) L) (FOREACH G (CDR X) L))))))
```

IV. FOREACH uses only pre-defined functions.

V. examples:

```
(APPLY FOREACH ((FUNCTION (LAMBDA (X L) (CONS L X))) (S N T) (I)))
(((I)) ((I)) ((I)))

(APPLY FOREACH ((FUNCTION (LAMBDA (X L) (ELEMENT ( L))) (U S A)
(T T T)                                                  (R U S S I A )))
```

(FOREACH* G X L)

I. FOREACH* applies a function of two arguments to the first element of the first given list, and to another list. This other list is the given function applied to the second element of the first given list and to another list similarly defined. The second argument in the case where the first argument is the last element of the first given list is the second given list.

II. FOREACH* is a function of:

G, a function of two arguments,

X, a list

L, a list.

III. definition:

```
(FOREACH* (LAMBDA (G X L) (COND
 ((NULL X) L)
 ((T) (G (CAR X) (FOREACH* G (CDR X) L))))))
```

IV. FOREACH* uses only pre-defined functions.

V. examples:

```
(APPLY FOREACH* ((FUNCTION (LAMBDA (X L) (CONS X L))) (A B C D) (E F G H)))
(A B C D E F G H)
```

(FOREACH** G X L)

I. FOREACH** writes a list whose first element is the given function of two arguments applied to the first element of the first given list, and to the second given list. Its second element is the given function applied to the second element of the first given list, and to the element described in the last sentence. Its third element is the function applied to the third element of the first given list, and to the element described in the previous sentence, and so on.

II. FOREACH** is a function of:

G, a function of two arguments,

X, a list,

L, a list.

III. definition:

```
(FOREACH** (LAMBDA (G X L) (COND
 ((NULL X) (LIST))
 ((T) (CONS (G CAR X) L) (FOREACH** G (CDR X) (G (CAR X) L)))))))
```

IV. FOREACH** uses only pre-defined functions.

V. examples:

```
(APPLY FOREACH** ((FUNCTION (LAMBDA (X L)
(EXPUNGE X L))) (A C D) (A B C D E F)))
((B C D E F) (B D E F) (B E F))

(APPLY FOREACH** ((FUNCTION (LAMBDA (X L)
(APPEND X L))) ((A) (B) (C) (D) (E)) ((BEEP))))
((A (BEEP)) (B A (BEEP)) (C B A (BEEP))
(D C B A (BEEP)) (E D C B A (BEEP)))
```

(PERMEATE G L)

I. PERMEATE applies a given function to each atomic symbol in a list. It then applies the function to each list of the resulting list, in one to one correspondence with the original list, which corresponds to a list which contained just atoms. It continues in this way until it applies the function to a list corresponding to the original list.

II. PERMEATE is a function of:

G, a function,

L, a list.

III. definition:

```
(PERMEATE (LAMBDA (G L) (COND
 ((ATOM L) (G L))
 ((T) (G (MAPCAR (FUNCTION (LAMBDA (L) (PERMEATE G L))) L))))))
```

IV. PERMEATE uses:

MAPCAR.

V. examples:

```
(APPLY PERMEATE ((FUNCTION (LAMBDA (L) (CONS L (QUOTE B)))) (B (B (B
                                                                  (A))))))

(((B) (((B) (((B) (((A)))))))))
```

(PERCOLATE G L)

I. PERCOLATE is defined like permeate, but starts by applying the function to lists of only atomic symbols.

II. PERCOLATE is a function of:

G, a function,

L, a list.

III. definition:

```
(PERCOLATE (LAMBDA (G L) (COND
 ((ATOM L) L)
 ((T) (G (MAPCAR (FUNCTION (LAMBDA (L) (PERCOLATE G L))) L))))))
```

IV. PERCOLATE uses:

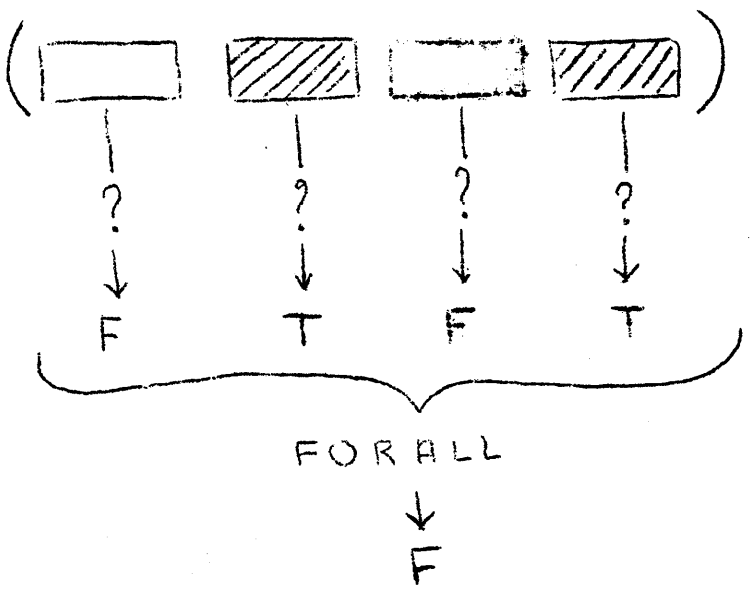
MAPCAR.

V. examples:

```
(APPLY PERCOLATE ((FUNCTION (LAMBDA (L) (APPEND L
(QUOTE B)))) (B (B (B (B (A)))))))
(B (B (B (B (A)))))
```

PREDICATES WHICH TEST A PREDICATE

ON VARIOUS MEMBERS OF A LIST



(FORALL L P)

I. FOR ALL takes the value T if the given predicate is true for all the elements of a given list, and F otherwise.

II. FORALL is a predicate of:

L, a list,

P, a predicate.

III. definition:

```
(FORALL (LAMBDA (P L)
 (OR (NULL L) (AND (P (CAR L)) (FORALL P (CDR L))))))
```

IV. FORALL uses only pre-defined functions.

V. examples:

```
(APPLY FORALL ((FUNCTION (LAMBDA (X) (ATOM X))) (A A A)))
T

(APPLY FORALL ((FUNCTION (LAMBDA (X) (ATOM X))) ((IC) B M)))
F
```

(FORSOME L P)

I. FORSOME takes the value T if the given predicate is true for any element of the given list, F otherwise.

II. FORSOME is a predicate of:

L, a list,

P, a predicate.

III. definition:

```
(FORSOME (LAMBDA (P L)
 (AND (NOT (NULL L)) (OR (P (CAR L)) (FORSOME P (CDR L))))))
```

IV. FORSOME uses only pre-defined functions.

V. examples:

```
(APPLY FORSOME ((FUNCTION (LAMBDA (X) (ATOM X))) (A (A A))))
T

(APPLY FORSOME ((FUNCTION (LAMBDA (X) (ATOM X))) ((A) (A) (A))))
F
```

(FORODD L P)

I. FORODD takes the value true if a given predicate is true for an odd number of elements of a given list.

II. FORODD is a predicate if:

L, a list of lists,

P, a predicate.

III. definition:

```
(FORODD (LAMBDA (P L)
 (AND (NOT (NULL L)) (OR (AND (P (CAR L)) (NOT (FORODD P
         (CDR L)))) (AND (NOT (P (CAR L))) (FORODD P
         (CDR L)))))))
```

IV. FORODD uses only pre-defined functions.

V. examples:

```
(APPLY FORODD ((FUNCTION (LAMBDA (X) (ATOM X))) (C (ON) F (US) I (NG))))
T

(APPLY FORODD ((FUNCTION (LAMBDA (X) (ATOM X))) ((CO) N (FU) S (ED))))
F
```

(SUCHTHAT* L P A1 A2)

I. SUCHTHAT is a function which takes a given value (or applies a given function) if there is an element on the given list such that the given property is true. Otherwise it takes the second given value (or function).

II. SUCHTHAT is a function of:

L, a list of lists,

P, a predicate,

A1, a function, atom, or list,

A2, a function, atom, or list.

III. definition:

```
(SUCHTHAT* (LAMBDA (L P A1 A2) (COND
 ((NULL L) A1)
 ((P (CAR L)) A2)
((T) (SUCHTHAT* (CDR L) P A1 A2)))))
```

IV. SUCHTHAT uses only pre-defined functions.

V. examples:

```
(APPLY SUCHTHAT* (((A) (B) C (D)) (FUNCTION (LAMBDA (S)
(ATOM S))) (NOPE) (YUP)))
(YUP)
```

(SUCHTHAT** L P PUNCTION)

I. SUCHTHAT** takes the value of (PUNCTION L) if there exists an element on the given list such that a given property is true; otherwise it is a null list.

II. SUCHTHAT** is a function of:

L, a list of lists,

P, a predicate,

PUNCTION, a function.

III. definition:

```
((SUCHTHAT** (LAMBDA (L P PUNCTION) (COND
 ((NULL L) (LIST))
 ((P (CAR L)) (PUNCTION L))
 ((T) (SUCHTHAT** (CDR L) P PUNCTION)))))
```

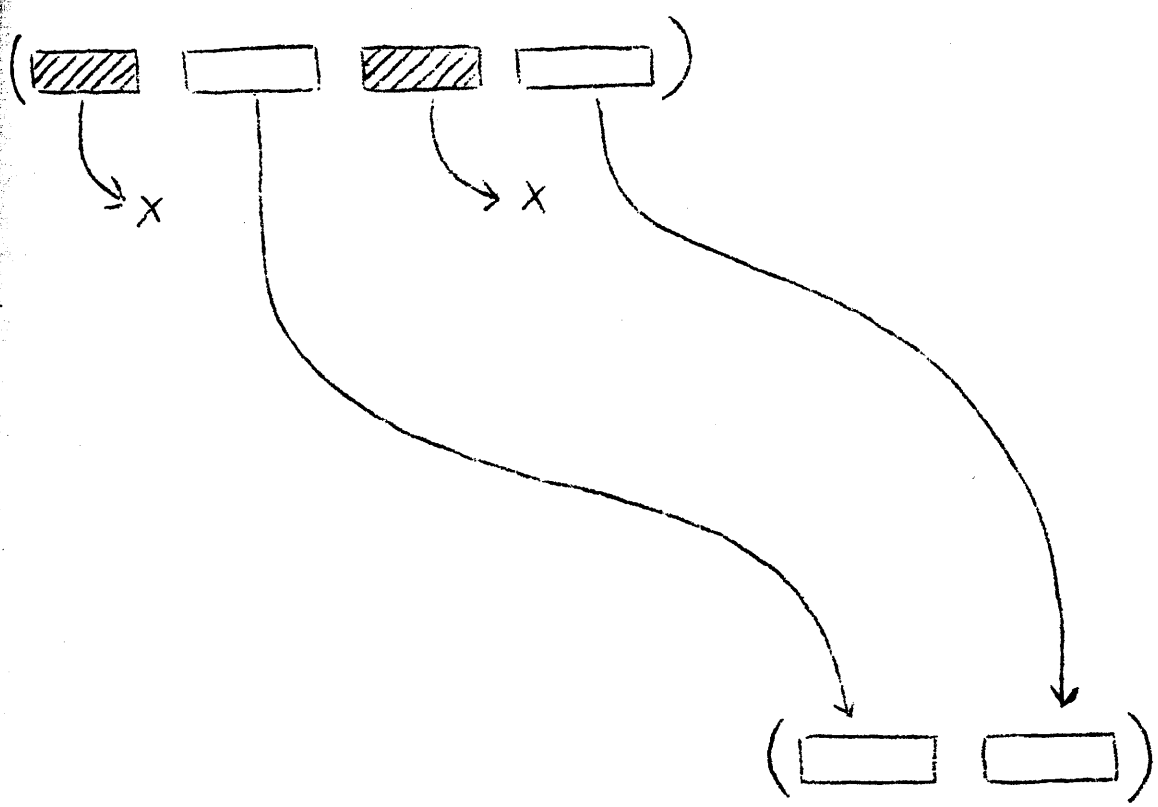
IV. SUCHTHAT** uses only pre-defined functions.

V. examples:

```
(APPLY SUCHTHAT** (((A B C D) (A B)) (FUNCTION (LAMBDA (S)
(ULTRATRIPLET S))) (FUNCTION (LAMBDA (S) (MAPCAR (FUNCTION
(LAMBDA (X) (COND ((NOT (ULTRATRIPLET X)) X) ((T) (LIST (CAR X)
(CADR X) (CADDR X)))))) S)))))
```

FUNCTIONS WHICH LIST

CERTAIN ELEMENTS OF A LIST...



(SUCHTHAT P L)

I. SUCHTHAT lists the first element in a given list for which a given predicate is true.

II. SUCHTHAT is a function of

L, a list,

P, a predicate.

III. definition:

```
(SUCHTHAT (LAMBDA (P L) (COND
 ((NULL L) (LIST))
 ((P (CAR L)) (CAR L))
 ((T) (SUCHTHAT P) (CDR L)))))
```

IV. SUCHTHAT uses only pre-defined functions.

V. examples:

```
(APPLY SUCHTHAT ((FUNCTION (LAMBDA (L) (ATOM L))) (A (B C) D)))
A
```

(POSSESSING P L)

I. POSSESSING writes a list of all the elements of a given list for which a given predicate is true.

II. POSSESSING is a function of:

P, a predicate,

L, a list.

III. definition:

```
(POSSESSING (LAMBDA (P L) (COND
 ((NULL L) (LIST))
 ((P (CAR L)) (CONS (CAR L) (POSSESSING P (CDR L))))
 ((T) (POSSESSING P (CDR L))))))
```

IV. POSSESSING uses only pre-defined functions.

V. examples:

```
(APPLY POSSESSING ((FUNCTION (LAMBDA (L) (ATOM L))) (P (O S) (E S S)
                                                        I N G)))
(P I N G)

(APPLY POSSESSING ((FUNCTION (LAMBDA (L) (ATOM L))) (A (B C) D)))
(A D)
```

(REMOVE I L)

I. REMOVE writes a list of all the elements of a given list except the first occurrence of a given element.

II. REMOVE is a function of:

I, a list element,

L, a list.

III. definition:

```
(REMOVE (LAMBDA (I L) (COND
 ((NULL L) (LIST))
 ((EQUAL I (CAR L)) (CDR L))
 ((T) (CONS (CAR L) (REMOVE I (CDR L)))))))
```

IV, REMOVE uses only pre-defined functions.

V. examples:

```
(APPLY REMOVE ((B) ((B) (A) (L) (E))))
((A) (L) (E))

(APPLY REMOVE (B (B A L E)))
(A L E)
```

(EXPUNGE X L)

I. EXPUNGE writes a list of all the elements on a given list except all those equal to a given element.

II. EXPUNGE is a function of:

X, a list element,

L, a list.

III. definition:

```
(EXPUNGE (LAMBDA (X L) (COND
 ((NULL L) (LIST))
 ((EQUAL X (CAR L)) (EXPUNGE X (CDR L)))
 ((T) (CONS (CAR L) (EXPUNGE X (CDR L)))))))
```

IV. EXPUNGE uses only pre-defined functions.

V. examples:

```
(APPLY EXPUNGE (I (M I S S I S S I P P I)))
(M S S S S P P)
```

(ALTERNATE L)

I. ALTERNATE makes a list of all the odd elements of a given list.

II. ALTERNATE is a function of:

L, a list.

III. definition:

```
(ALTERNATE (LAMBDA (L) (COND
 ((NULL L) (LIST))
 ((NULL (CDR L)) (LIST))
 ((T) (CONS (CAR L) (ALTERNATE (CDDR L)))))))
```

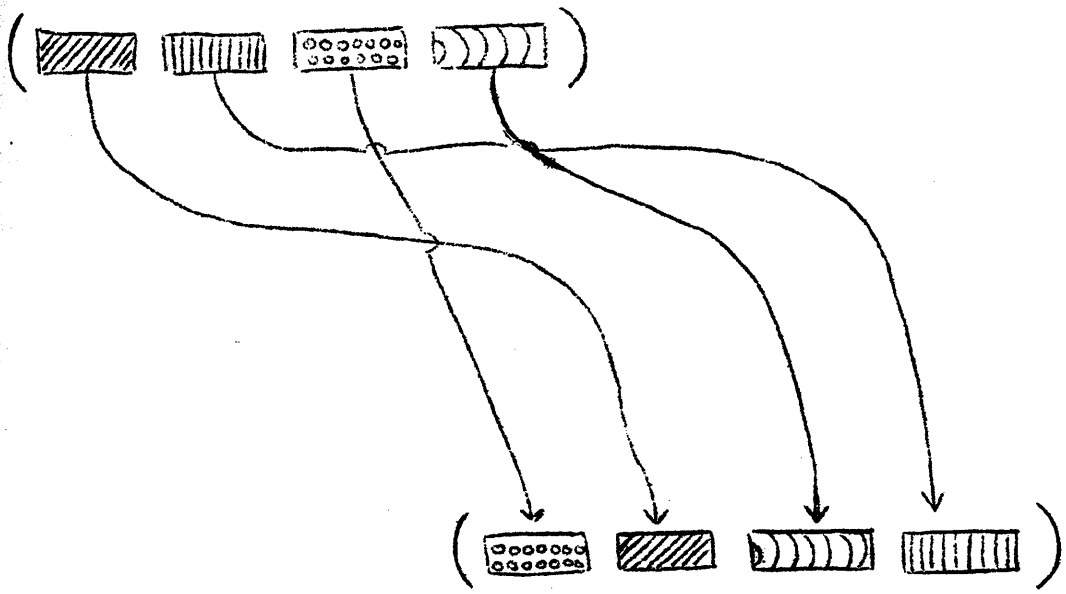
IV. ALTERNATE uses only pre-defined functions.

V. examples:

```
(APPLY ALTERNATE ((U S S R)))
(U S)

(APPLY ALTERNATE ((A A B B C C)))
(A B C)
```

FUNCTIONS WHICH DO

VARIOUS MANIPULATIONS...



(REVERSE L)

I. REVERSE rewrites a given list with the order of the elements reversed.

II. REVERSE is a function of:

L, a list.

III. definition:

```
(REVERSE (LAMBDA (L) (REVERSE* L (LIST))))
(REVERSE* (LAMBDA (L M) (COND
 ((NULL L) M)

 ((T) (REVERSE* (CDR L) (CONS (CAR L) M))))))
```

IV. REVERSE uses:

The auxiliary function REVERSE*

V. examples:

```
(APPLY REVERSE ((S E R U T A N)))
(N A T U R E S)

(APPLY REVERSE ((E M B A R G O S)))
(S O G R A B M E)
```

(COLLECT L)

I. COLLECT regroups the elements of a list so that all reoccurrences of each element, if any, are made immediately after the first occurrence.

II. COLLECT is a function of:

L, a list of any number of elements.

III. definition:

```
(COLLECT (LAMBDA (L) (COND
 ((NULL L) (LIST))
 ((ELEMENT (CAR L) (CDR L)) (CONS (CAR L) (COLLECT
  (CONS (CAR L) (REMOVE (CAR L) (CDR L))))))
 ((T) (CONS (CAR L) (COLLECT (CDR L)))))))
```

IV. COLLECT uses:

ELEMENT, REMOVE.

V. examples:

```
(APPLY COLLECT ((N I X O N)))
(N N I X O)
```

(REALTERNATE L M)

I. REALTERNATE places the given list element after each element of a given list.

II. REALTERNATE is a function of:

L, a list,

M, a list element.

III. definition:

```
(REALTERNATE (LAMBDA (L M) (COND
 ((NULL L) (LIST))
 ((NULL (CDR L)) L)
 ((T) (CONS (CAR L) (CONS M (REALTERNATE (CDR L) M))))))
```

IV. REALTERNATE uses only pre-defined functions.

V. examples:

```
(APPLY REALTERNATE ((B N N S) A))
(B A N A N A S)

(APPLY REALTERNATE ((A B C) (1)))
(A (1) B (1) C)
```

(APPEND A B)

I. APPEND makes a list of the elements of two lists.

II. APPEND is a function of:

A, a list,

B, a list.

III. definition:

```
(APPEND (LAMBDA (A B) (COND
 ((NULL A) B)
 ((T) (CONS (CAR A) (APPEND (CDR A) B))))))
```

IV. APPEND uses only pre-defined functions.

V. examples:

```
(APPLY APPEND ((D O G) (H O U S E)))
(D O G H O U S E)
```

(SUBSTITUTE X Y L)

I. SUBSTITUTE replaces every occurrence of one given element on a given list by a second given element.

II. SUBSTITUTE is a function of:

X, a list element

Y, a list element

L, a list.

III. definition:

```
(SUBSTITUTE (LAMBDA (X Y L) (COND
 ((NULL L) (LIST))
 ((EQUAL Y (CAR L)) (CONS X (SUBSTITUTE X Y (CDR L))))
 ((T) (CONS (CAR L) (SUBSTITUTE X Y (CDR L)))))))
```

IV. SUBSTITUTE uses:

EQUAL.

V. examples:

```
(APPLY SUBSTITUTE (B A (A (A (A (A))))))
(B (A (A (A))))

(APPLY SUBSTITUTE (B A (A A A)))
(B B B)
```

(EXTRACT P L)

I. EXTRACT makes a list of two elements, The first is composed of all the elements of a given list with a given property; the second is composed of all the remaining elements.

II. EXTRACT is a function of:

P, a predicate,

L, a list.

III. definition:

```
(EXTRACT (LAMBDA (P L) (COND
 ((NULL L) (LIST (LIST) (LIST))
 ((P (CAR L)) ((LAMBDA (X) (CONS (CONS (CAR L) (CAR X))
  (CDR X))) (EXTRACT P (CDR L))))
 ((T) ((LAMBDA (X) (LIST (CAR X) (CONS (CAR L) (CADR X))))
  (EXTRACT P (CDR L)))))))
```

IV. EXTRACT uses only pre-defined functions.

V. examples:

```
(APPLY EXTRACT ((LAMBDA (L) (SINGLET L)) (B (O) O M (A) (T) (O) (M) B)))
(((O) (A) (T) (O) (M)) (B O M B))
```

(AMONG X L)

I. AMONG adds the given list element to the given list if it is not on the list.

II. AMONG is a function of:

X, a list element

L, a list.

III. definition:

```
(AMONG (LAMBDA (X L) (COND
 ((NULL L) (LIST X))
 ((EQUAL X (CAR L)) L)
 ((T) (CONS (CAR L) (AMONG X (CDR L)))))))
```

IV. AMONG uses:

EQUAL.

V. examples:

```
(APPLY AMONG (K (D A R)))
(D A R K)

(APPLY AMONG (B (B I T)))
(B I T)
```

ARITHMETIC FUNCTIONS.....

$1 + 1 = 0_1$

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 1 = 0_1$

(BINARY L)

I. BINARY writes the binary equivalent of a given octal number expressed as a list of its (atomic) digits.

II. BINARY is a function of:

L, a list of octal digits.

III. definition:

```
(BINARY (LAMBDA (L) (COND
 ((NULL L) (LIST))
 ((EQUAL (CAR L) (QUOTE 7)) (CONS (QUOTE 1) (CONS (QUOTE 1)
  (CONS (QUOTE 1) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 6)) (CONS (QUOTE 1) (CONS (QUOTE 1)
  (CONS (QUOTE 0) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 5)) (CONS (QUOTE 1) (CONS (QUOTE 0)
  (CONS (QUOTE 1) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 4)) (CONS (QUOTE 1) (CONS (QUOTE 0)
  (CONS (QUOTE 0) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 3)) (CONS (QUOTE 0) (CONS (QUOTE 1)
  (CONS (QUOTE 1) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 2)) (CONS (QUOTE 0) (CONS (QUOTE 1)
  (CONS (QUOTE 0) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 1)) (CONS (QUOTE 0) (CONS (QUOTE 0)
  (CONS (QUOTE 1) (BINARY (CDR L))))))
 ((EQUAL (CAR L) (QUOTE 0)) (CONS (QUOTE 0) (CONS (QUOTE 0)
  (CONS (QUOTE 0) (BINARY (CDR L))))))
```

IV. BINARY uses only pre-defined functions.

V. examples:

```
(APPLY BINARY ((7)))
(1 1 1)

(APPLY BINARY ((6 3)))
(1 1 0 0 1 1)
```

(COUNT L)

I. COUNT writes a list of ones equal in number to the numerical value of the first given list, a binary number in list form.

II. COUNT is a function of:

L, a list whose elements are the (atomic) digits of a binary number,

C, a list element.

III. definition:

```
(COUNT (LAMBDA (L)
  (COUNT* (REVERSE L) (LIST (QUOTE 2))))))
(COUNT* (LAMBDA (A B) (COND
 ((NULL A) (LIST))
 ((EQ (CAR A) (QUOTE 1)) (APPEND B (COUNT*
  (CDR A) (APPEND B B))))
 ((T) (COUNT* (CDR A) (APPEND B B))))))
```

IV. COUNT uses:

APPEND, REVERSE, and the auxiliary found in COUNT*.

V. examples:

```
(APPLY COUNT ((1 1 0 1)))
(1 1 1 1 1 1 1 1 1 1 1 1 1)
```

(ENGLISH L)

I. ENGLISH rewrites a list so that the first element of the given list is placed after each of the remaining elements of the list.

II. ENGLISH is a function of:

L, a list.

III. definition:

```
(ENGLISH (LAMBDA (L)
 (REALTERNATE (CDR L) (CAR L))))
```

IV. ENGLISH uses:

REALTERNATE

V. examples:

```
(APPLY ENGLISH ((- A B C)))
(A - B - C)

(APPLY ENGLISH ((A B N N S)))
(B A N A N A S)
```

(POLISH L)

I. POLISH takes an algebraic expression (of only one operation) in list form and converts it to Polish notation.

II. POLISH is a function of:

L, a list of the (atomic) symbols of an algebraic expression of one operation.

III. definition:

```
(POLISH (LAMBDA (L) (COND
 ((AND (ULTRATRIPLET L) (ODDPLET L) (UNIFORM L (CADR L)))
  (CONS (CADR L) (ALTERNATE L)))
 ((T) L))))
```

IV. POLISH uses:

ULTRATRIPLET, ODDPLET, UNIFORM, ALTERNATE.

V. examples:

```
(APPLY POLISH ((A - B * C - D - E)))
(A - B * C - D - E)

(APPLY POLISH ((A - B - C - D - E)))
(- A B C D)
```

(POSPOLISH L)

I. POSTPOLISH changes an algebraic expression (of one operation) to postpolish form.

II. POSTPOLISH is a function of:

L, a list of any length with an odd number of terms, and whose even terms are identical. (if the argument is not of this form, (POSTPOLISH L) is equal to L.)

III. definition:

```
(POSTPOLISH (LAMBDA (L) (COND
 ((AND (ULTRATRIPLET L) (ODDPLET L) (UNIFORM L (CADR L)))
  (APPEND (ALTERNATE L) (LIST (CADR L))))
 ((T) L))))
```

IV. POSTPOLISH uses:

ULTRATRIPLET, ODDPLET, UNIFORM, APPEND, ALTERNATE.

V. examples:

```
(APPLY POSTPOLISH ((A - B - C - D - E)))
(A B C D -)

(APPLY POSTPOLISH ((A - B - C - D * E)))
(A - B - C - D * E)
```

(INTEGERS)

I. INTEGERS is equal to a list of the integers from 1 to 60.

II. INTEGERS is a function of no arguments.

III. definition:

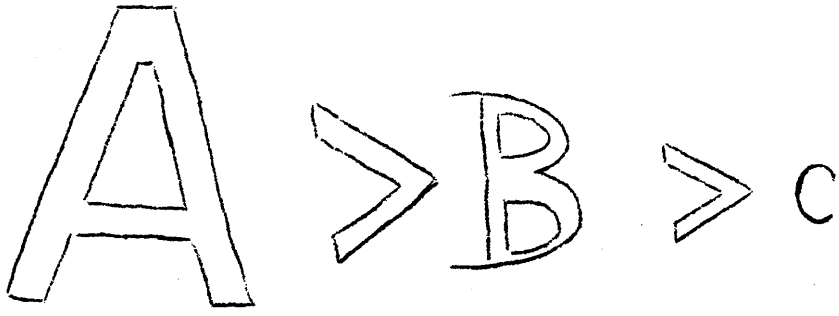
```
(INTEGERS (LAMBDA ( )
 (QUOTE (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44
 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60)))))
```

IV. INTEGERS uses only pre-defined functions.

V. examples:

```
(APPLY INTEGERS ())
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60)
```

ORDER.....,



(RANK L O)

I. RANK rewrites a given list so that its elements are ordered according to a second list.

II. RANK is a function of:

L, a list,

O, a list.

III. definition:

```
(RANK (LAMBDA (L O) (COND
 ((NULL L) (LIST))
 ((T) (CONS (MINIMUM L O) (RANK (REMOVE (MINIMUM L O)) L) O))))))
```

IV. RANK uses:

MINIMUM, REMOVE.

V. examples:

```
(APPLY RANK ((N E S T) (S O L E M N I T Y)))
(GC)
((S) (E) (N) (T))

(APPLY RANK ((A L E) (S O L E M N I T Y)))
((L) (E) (A))
```

(ORDER X Y L)

I. ORDER takes the value T if the first given element is on the given list, and the second is not; or if both are on the list, and the first comes before the second; F otherwise.

II. ORDER is a function of:

X, a list element,

Y, a list element,

L, a list.

III. definition:

```
(ORDER (LAMBDA (X Y L) (COND
 ((NULL L) F)
 ((EQUAL X (CAR L)) T)
 ((EQUAL Y (CAR L)) F)
 ((T) (ORDER X Y (CDR L))))))
```

IV. ORDER uses:

EQUAL.

V. examples:

```
(APPLY ORDER (B A (A L E)))
F

(APPLY ORDER (A B (A L E)))
T

(APPLY ORDER (A B (B A L E)))
F
```

(MINIMUM L O)

I. MINIMUM selects the smallest element on a list according to a second given list.

II. MINIMUM is a function of:

L, a list,

O, a list.

III. definition:

```
(MINIMUM (LAMBDA (L O) (COND
 ((NULL L) (LIST))
 ((NULL (CDR L)) L)
 ((ORDER (CAR L) (CADR L) O) (MINIMUM (CONS (CAR L) (CDDR L)) O))
 ((T) (MINIMUM (CDR L) O)))))
```

IV. MINIMUM uses:

ORDER,

V. examples:

```
(APPLY MINIMUM ((A C E) (A B C D E)))
(A)

(APPLY MINIMUM ((F G H) (A B C D E)))
(H)
```

(LEXORDER X Y L)

I. LEXORDER takes the value T if, for the first pair of non-equal corresponding elements in two given lists, the element of the first list precedes the corresponding element of the second list in the corresponding element of the third list; otherwise F.

II. LEXORDER is a predicate of:

X, a list element,

Y, a list element,

L, a list.

III. definition:

```
(LEXORDER (LAMBDA (X Y L) (COND
 ((NULL L) F)
 ((NOT (EQUAL (CAR X) (CAR Y))) (ORDER (CAR X) (CAR Y) (CAR L)))
 ((T) (LEXORDER (CDR X) (CDR Y) (CDR L))))))
```

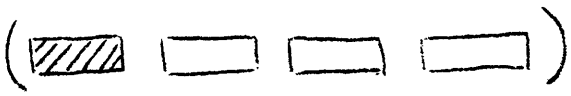
IV. LEXORDER uses:

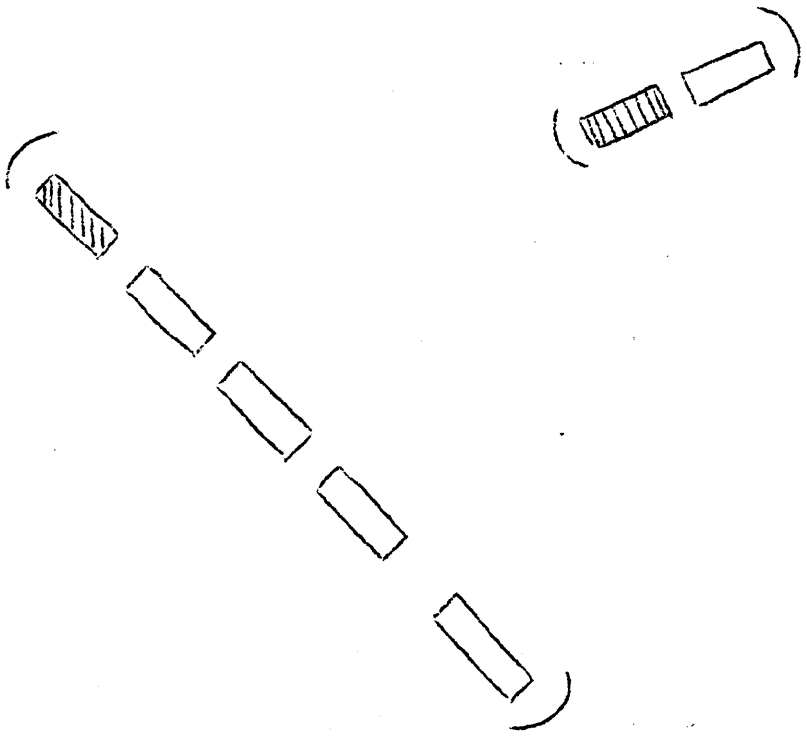
ORDER.

V. examples:

```
(APPLY LEXORDER ((A B C) (A B D) ((P Q R) (S T U) (A B C D))))
T
```

MISCELLANEOUS FUNCTIONS.....





(ACCUMULATE L M)

I. ACCUMULATE will make a list of distinct items from two given lists, if the second list is of distinct items.

II. ACCUMULATE is a function of:

L, a list,

M, a list of distinct elements.

III. definition:

```
(ACCUMULATE (LAMBDA (L M) (COND
 ((NULL L) M)
 ((ELEMENT (CAR L) M) (ACCUMULATE (CDR L) M))
 ((T) (ACCUMULATE (EXPUNGE (CAR L) (CDR L)) (CONS (CAR L) M))))))
```

IV. ACCUMULATE uses:

ELEMENT, EXPUNGE.

V. examples:

```
(APPLY ACCUMULATE ((A L A B A M A) (A B C D)))
(M L A B C D)

(APPLY ACCUMULATE ((A A B C D) (B C D)))
(A B C D)
```

(COMPOSITE A B S)

I. COMPOSITE makes a list of all doublets (X Z) such that there exists a doublet (X Y) on the first given list, there exists a doublet (X Z) on the second given list, and X, Y, Z are all on the third given list.

II. COMPOSITE is a function of:

A, a list of doublets,

B, a list of doublets,

S, a list.

III. definition:

```
(COMPOSITE (LAMBDA (A B S)
 (POSSESSING (FUNCTION (LAMBDA (U) (FORSOME (FUNCTION (LAMBDA (V)
  (AND (ELEMENT (LIST (CAR U) V) A) (ELEMENT
  (LIST V (CADR U)) B)))) S))) (CARTESIAN (LIST S S)))))
```

IV. COMPOSITE uses:

POSSESSING, FORSOME, ELEMENT, CARTESIAN.

V. examples:

```
(APPLY COMPOSITE (((A B) (C D) (E F)) ((D A) (B E) (F C)) (A B C D E)))
((A E) (C A))
```

(REPEAT X I)

I. REPEAT makes a list of the first given list repeated as many times as there are elements on the second given list.

II. REPEAT is a function of:

X, a list,

I, a list.

III. definition:

```
(REPEAT (LAMBDA (X I) (COND
 ((NULL I) LIST))
 ((T) (CONS X (REPEAT X (CDR I)))))))
```

IV. REPEAT uses only pre-defined functions.

V. examples:

```
(APPLY REPEAT ((H E L P) (A B C D)))
((H E L P) (H E L P) (H E L P) (H E L P))
```

(TALLYCOPY L M)

I. TALLYCOPY lists as many elements from the second given list, starting from the left, as there are elements on the first given list.

II. TALLYCOPY is a function of:

L, a list,

M, a list with at least as many elements as L.

III. definition:

```
(TALLYCOPY (LAMBDA (L M) (COND
 ((NULL L) (LIST))
 ((T) (CONS (CAR M) (TALLYCOPY (CDR L) (CDR M)))))))
```

IV. TALLYCOPY uses only pre-defined functions.

V. examples:

```
(APPLY TALLYCOPY ((D A R K) (K K K)))
(K K K)
```

(TALLYCOMPLEMENT L M)

I. TALLYCOMPLEMENT lists the elements of the second given list with as many elements removed, starting from the left, as there are elements on the first given list.

II. TALLYCOMPLEMENT is a function of:

L, a list,

M, a list of at least as many elements as L.

III. definition:

```
(TALLYCOMPLEMENT (LAMBDA (L M) (COND
 ((NULL L) M)
 ((T) (TALLYCOMPLEMENT (CDR L) (CDR M))))))
```

IV. TALLYCOMPLEMENT uses only pre-defined functions.

V. examples:

```
(APPLY TALLYCOMPLEMENT ((D A R K) (R A I N)))
()
```

(FRAGMENT N L)

I. FRAGMENT lists the elements of the given list up to the first occurrence of the given list element.

II. FRAGMENT is a function of:

N, a list element,

L, a list.

III. definition:

```
(FRAGMENT (LAMBDA (N L) (COND
 ((EQ N (CAR L)) (LIST))
 ((T) (CONS (CAR L) (FRAGMENT N (CDR L)))))))
```

IV. FRAGMENT uses only pre-defined functions.

V. examples:

```
(APPLY FRAGMENT (P (E G G P L A N T)))
(E G G)
```

(CARTESIAN* A B X)

I. CARTESIAN* forms a list of all possible doublets whose: first element is CAR of A, and whose second element is an element of B; first element is an element of CDR of A, and whose second element is an element of X.

II. CARTESIAN* is a function of:

A, a list,

B, a list,

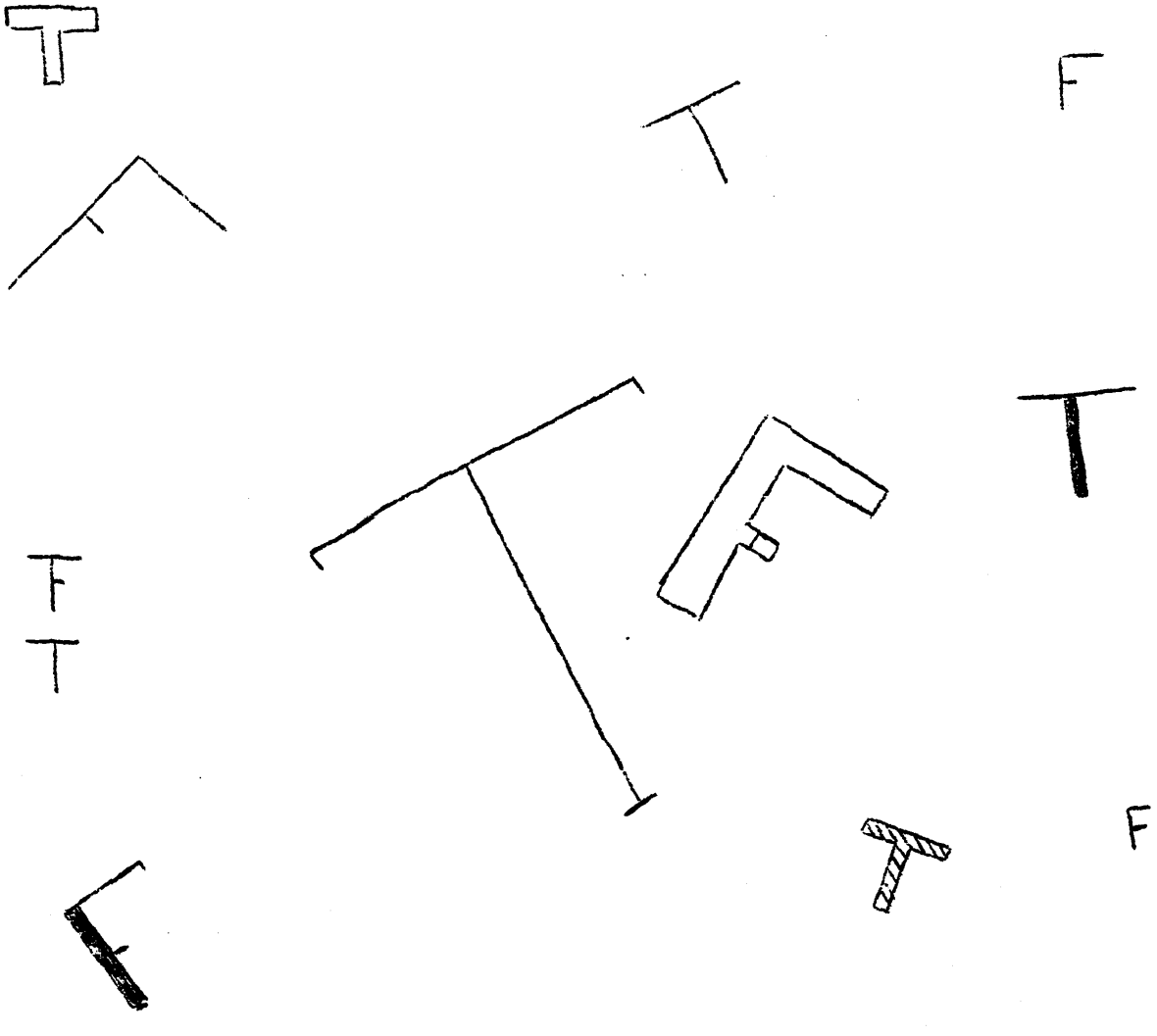
X, a list.

III. definition:

```
(CARTESIAN* (LAMBDA (A B X) (COND
 ((NULL A) (LIST))
 ((NULL B) (CARTESIAN* (CDR A) X X))
 ((T) (CONS (CONS (CAR A) (CAR B)) (CARTESIAN* A (CDR B) X))))))
```

IV. CARTESIAN* uses only pre-defined functions.

MISCELLANEOUS PREDICATES.....



(EQUIVALENT X Y L)

I. EQUIVALENT takes the value T if the two given elements are contained in the same sub-list of the given list; otherwise F.

II. EQUIVALENT is a predicate of:

X, a list element

Y, a list element,

L, a list.

III. definition:

```
(EQUIVALENT (LAMBDA (X Y L) (COND
 ((NULL L) F)
 ((ELEMENT X (CAR L)) (ELEMENT Y (CAR L)))
 ((T) (EQUIVALENT X Y (CDR L))))))
```

IV. EQUIVALENT uses:

ELEMENT.

V. examples:

```
(APPLY EQUIVALENT ((A) (B) ((A C) (B D) (B C) (A B))))
F
```

(UNIFORM L M)

I. UNIFORM takes the value T if all the even terms of the given list are the same as a given element; otherwise F.

II. UNIFORM is a predicate of:

L, a list of at least three, and an odd number of elements,

M, a list element.

III. definition:

```
(UNIFORM (LAMBDA (L M)
 (OR (AND (TRIPLET L) (EQUAL (CADR L) M))
  (AND (ULTRATRIPLET L) (EQUAL (CADR L) M) (UNIFORM (CDDR L) M)))))
```

IV. UNIFORM uses:

TRIPLET, EQUAL, ULTRATRIPLET.

V. examples:

```
(APPLY UNIFORM ((A - B - C) (-)))
F

(APPLY UNIFORM ((A - B - C - D) -))
T
```

(EQUAL X Y)

I. EQUAL takes the value T if the two given lists are identical.

II. EQUAL is a predicate of:

X, a list,

Y, a list.

III. definition:

```
(EQUAL (LAMBDA (X Y)
 (OR (EQ X Y) (AND (NULL X) (NULL Y)) (AND (NOT
  (OR (NULL X) (NULL Y) (ATOM X) (ATOM Y))) (EQUAL (CDR X) (CDR Y))
  (EQUAL (CAR X) (CAR Y))))))
```

IV. EQUAL uses only pre-defined functions.

V. examples:

```
(APPLY EQUAL ((A B C D) (A B C D)))
T

(APPLY EQUAL (A A))
T
```

(SIMILAR X L)

I. SIMILAR--two lists or list elements are SIMILAR if: they are both null; they are identical; the first is the atom---or corresponding elements up to the atom--in the first are similar.

II. SIMILAR is a predicate of:

X, a list,

L, a list.

III. definition:

```
(SIMILAR (LAMBDA (X L)
 (OR
  (EQ X (QUOTE -))
  (EQ X L)
  (AND (NULL X) (NULL L))
  (AND
   (NOT (ATOM X)) (NOT (NULL X))
   (OR (AND (EQ (CAR X) (QUOTE ---)) (OR (NULL L) (NOT (ATOM L))))
    (AND (NOT (ATOM L)) (NOT (NULL L) (SIMILAR (CDR X) (CDR L))
    (SIMILAR (CAR X) (CAR L)))))))))
```

IV. SIMILAR uses only pre-defined functions.

V. examples:

```
(APPLY SIMILAR (A A))
T

(APPLY SIMILAR (() ()))
T

(APPLY SIMILAR ((A (B (C D) E) (F G)) (A (B (C D) E) (F G))))
T

(APPLY SIMILAR ((A (B (C D) E) (F G)) (A (B (C D) E) (F G))))
F
```

INDEX OF DE-BUGGED FUNCTIONS