DLU 75/9

LISP - details

INTERLISP/360-370

by

Anders Haraldson

UPPSALA UNIVERSITY
Datalogilaboratoriet
Department of Computer Sciences

LISP-details

INTERLISP/360-370

by Anders Haraldson

ABSTRACT

This paper gives a tutorial introduction to INTERLISP/360-370, a subset of INTERLISP, which can be implemented on IBM/360 and similar systems. Descriptions of a large number of functions in INTERLISP with numerous examples, exercises and solutions are contained. The use of edit, break, advice, file handling and compiler are given and both interactive and batch use of the system is taken care of.

UPPSALA UNIVERSITET

datalogilaboratoriet

department of computer science

Printed in Sweden at Uppsala University
ISBN 91-506-0034-6

Contents

Preface

INTERLISP/360-370[1] is an implementation of a subset of INTERLISP (ref 1) on the IBM/360 and similar systems. It contains the interpreter, the set of basic functions, edit, break, advise and the compiler. Other packages such as DWIM, the Programmer's Assistent, CLISP and QLISP can quite easily extend the system.

This paper gives a tutorial introduction to the system. For those interested only in gaining some basic ideas about LISP there is an informal introduction by Sandewall (ref 3). No previous knowledge about LISP is required to read this text, but an elementary course in computer and programming use can make its understanding easier. There are a lot of differences between INTERLISP and LISP 1.5 (ref 4). The reader already having a knowledge of LISP 1.5 will find it advantageou to find out the differences for himself, as it is not normally pointed out where the differences appear. Some minor differences between INTERLISP/360-370 and INTERLISP occur, mostly machine-dependent, such as character sets, OS-interface etc.

The paper describes both the interactive and batch use of the system. Implementation-dependent features, such as control cards (commands) for running the system and data sets needed for file handling are included in an appendix.

There are a lot of references to the "LISP manual", by this is meant the INTERLISP/360-370 Reference Manual (ref 2). The term LISP is also used as an abbrevation of INTERLISP and what is said about LISP in this text is not necessarily legal in other LISP dialects.

In each section the fact is clarified with examples, and at the end of each section there are a number of exercises. Solutions can be found following the section part. Many of the examples and exercises are given to define system functions, showing how they work and familiarising the reader with them. These functions are *-marked; for <u>put</u> the function name <u>*put</u> is used. It is very important to use this *-marked version of the

[1] Information about INTERLISP/360-370 can be obtained from UDAC, Box 2103, S-750 02 Uppsala, Sweden, or Datalogilaboratoriet, Sturegatan 1, S-752 23 Uppsala, Sweden.

function if it is to be tested on the computer, as otherwise there is a great chance to erroniously redefine system-functions. The definitions shown in this paper on the system-functions are in most cases not identical with the way they are really implemented. In many cases the definitions given are not the best but they are forwarded for pedagogical reasons and in some cases the solutions are only partial, they do not take care of all the different cases which can occur etc.

The INTERLISP system contains hundereds of system functions and I think it meaningless to read this text with the intention of learning each and every function by heart. Better is to read quickly through and look at the examples than try to understand its full definition and later when you have found the need for a special function go back again and study it again. If you have an interactive system available it is simple to check how a system function works, which sometimes is faster than to understand its definition. Many functions are not described fully in this paper; their full definition is found in the reference manual.

In the first section we use M-notation, where functions, variables and conditional expressions are written in small letters (lower case), and S-expressions in capitals. Later we mostly use the S-notation (the notation used when communicating with the computer), the M-notation is only used for describing simple forms such as

```
     car[(A B)], car[1], memb[car[1],foo[cdr[1]]]
```

This paper is not in its final version, so I would appreciate comments and suggestions about it. Although all examples and solutions are checked out by computer there will still be errors in them.

Acknowledgements

Thanks to Mats Nordström, Erik Sandewall, Jaak Urmi, other members of Datalogilaboratoriet, students and others for reading this manuscript and giving valuable comments and criticism.

Thanks also to Mark Payter and Lis Stafberg for typing and editorial help.

References

1. Warren Teitelman
   INTERLISP REFERENCE MANUAL
   XEROX, Palo Alto Research Center,
   3180 Proter Drive, Palo Alto, Calif 94304, USA, 9174

2. INTERLISP/360 and 370 REFERENCE MANUAL
   Uppsala Datacentral
   Box 2103, S-750 02 Uppsala, Sweden, 1974

3. Erik Sandewall
   LISP: Principles
   Datalogilaboratoriet
   Sturegatan 1, S-752 23 Uppsala, Sweden, 197?

4. John McCarthy et al
   LISP 1.5 Programmer's Manual
   The M.I.T. Press, Cambridge, Mass, USA, 1962

5. Clark Weissman
   LISP 1.5 PRIMER
   Dickenson Publishing Company, Belmont, Calif, USA, 1967

6. Donald Knuth
   Fundamental Algorithms
   Addison Weley, USA, 1968

7. Erik Sandewall
   A proposed solution to the FUNARG problem
   Datalogilaboratoriet, Sturegatan 1, S-752 23 Uppsala Sweden, 1970

1. Primary datatypes in LISP

1.1 This first section will describe the syntax of atom, list, and string, which are the most primary datatypes in LISP. The different properties of these datatypes will be described later. The other datatypes will also be defined later - a table containing all datatypes can be found in the LISP manual.[1]

1.2 The characters are separated in two groups

- delimiters (space), ↲ (end of line), % (escape), (,), >, <, ", '
- non-delimiters the remaining characters. The character set can depend on the type of terminal, but normally includes both lower and upper case.

1.3 Literal atom.[2] A sequence of non-delimiters, that cannot be interpreted as a number.

eg ABCD NIL $A12/ 123++45 VERYVERYLONGATOM

In this text we will use capital letters for literal atoms.

1.4 Number[2]

Integer. An optional sign (+ or -) and a sequence of decimal digits.

eg 1234 -12 +1111 0

---

[1] When refering to the LISP manual we intend reference to the INTERLISP/360-370 Reference Manual.

[2] Literal atoms and numbers are together called atoms. Throughout the text atom is used as an abbreviation of literal atom, when no misunderstanding can occur.

Floating-point number. An integer, followed by a decimal point, followed by a sequence of decimal digits - called the fraction - followed by an exponent, represented by E and an integer. The different parts can be omitted but there must remain sufficient parts to enable distinction from an integer.

```
eg     5.210E-10    12.     .123     10E2
```

1.5 There are possibilities of constructing atoms internally which contain delimiters. This can be done by packing the atom as shown in Section 22. If we wish to read an atom containing delimiters we can do so by preceding every delimiter by the *escape character* %.

```
eg     AB%(12%"%%  will internally be represented as the atom
       AB(12"%
```

The LISP read-routine treats the next character after the escape character as a non-delimiting character.

1.6 *List*

A *list* can be constructed by other LISP elements, eg atoms and other lists, enclosed by parentheses or brackets.

```
eg     (A B C),   ((THIS IS) A ((LIST) STRUCTURE)), (), ((())).
```

In LISP it often happens that we have a considerable number of parentheses following each other, as in

```
eg     (A (B (C (D (E)))))
```

In such cases we can use the right brackets ,>, for terminating the list.

```
       (A (B (C (D (E>
```

The general rule being that the right bracket matches either the nearest left bracket, <, or the beginning of the list. These are all equivalent lists

```
(A (B (C)) (D (E)) (F (G)))
(A (B (C)) (D (E)) (F (G>
(A <B (C> <D (E> (F (G>
(A (B (C)) (D (E)) <F (G>>
```

The list (A (B C) D) contains three *list elements*, the atom A, the list (B C) and the atom D. (B C) is called a *sublist* of the original list. The list contains two atoms A and D at the top level of the list, but four atoms at all levels.

The empty list can be represented both by () and NIL. Usually we use NIL. NIL can be interpreted both as a list and as an atom. Care must be shown here; observe by the different function-definitions how NIL is treated.

When the LISP print-routine prints a list it is not certain that it will print the list in the same way as we gave in the read-routine. The ordinary print-functions print the list with parentheses, but the pretty-print functions will also use brackets. The empty list is always printed as NIL.

1.7 String

A string is a " followed by a sequence of any character except " and % (escape character) terminated by a " .

```
eg    "THIS IS A STRING"    "((()   ()"
```

" and % can be included in the string by preceding them with the escape character %.

```
eg    "AB%"C%%"  is internally the string AB"C%
```

Exercises

1. Classify each of these expressions if it is correct, as either literal atom; integer; floating point number; list or string. We assume that the LISP read-routine will read them.

```
a.  ABC123
b.  123ABC
c.  1.23E+12
d.  (
e.  ((()))
f.  123(
g.  %(%)
h.  (A B (C D))
i.  +123
j.  12.+34
k.  ((((((1))))))
l.  "AB("
m.  AB"NIL"
n.  """
o.  "%"% %""
p.  (A , B , C , D)
q.  %%%
r.  ("A" "B")
s.  (((A B) C D) E)
t.  (A B <C D> <E>>
u.  <A B> )
v.  <A B)>
x.  (A <B <(C> )
y.  (A B . C)
z.  (A.B)
```

2. Representation of atoms and lists

2.1 Atoms and lists are internally represented as records and this section will show what information is stored in these records. For the exact internal representation such as the order of the fields in the records, the number of bits for each field etc, consult the LISP manual. We will use the records as a graphical representation of list structures.

2.2 In a language like LISP, where list structure is the most important data structure, there must be pointers (references). A pointer in INTERLISP/360-370 contains both the datatype number of the referenced data element and the address to that element. All datatypes and their associated numbers are to be found tabulated in the LISP manual.

data-type	address

2.3 A literal atom is a record called atom cell with four fields

- pname pointer, a pointer to an area where the atom's print-name is stored. The print-name of an atom is the sequence of characters which defines the atom.
- value cell, a pointer to the atom's global value if it exists, otherwise a pointer to the atom NOBIND. See further Section 15.
- property-list, a pointer to the atom's property-list if it exists, otherwise a pointer to NIL. See further Section 5.
- function cell, a pointer to the atom's function definition if it exists, otherwise a pointer to NIL. See further Section 14.

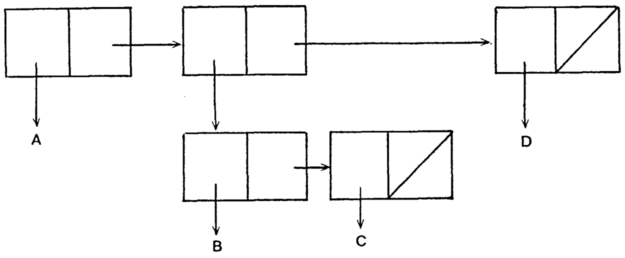
pname pointer	value cell	property list	function cell

An atom is unique, which means that for a given pname string - the sequence of characters defining an atom - there can only be one atom cell with that pname string. When the LISP read-routine reads an atom it first attempts to discover if there already exists an atom cell with that pname string and if it exists, use it, otherwise the read-routine creats a new atom cell for that atom.

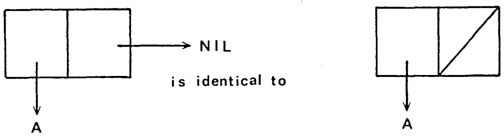
2.4 Numbers are represented in different ways. Integers are separated into small integers and big integers with two different representations. From the user's point-of-view there is no real difference. The small integers are unique, they are represented in the pointer, but all other numbers are not. Further information concerning the numerical atom's representation is to be found in the LISP manual.

2.5 A list is a chain of list cells, where a list cell is a record of two fields, both containing pointers. The first pointer references a list element and the second pointer references the rest of the list.

eg (A (B C) D)

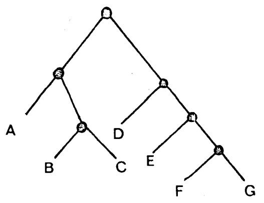


The pointers to the atoms A, B, C and D are the references to respective atom cell. It will be found convenient to write only the atom's print-name instead of drawing its record. A list normally ends with NIL and the slanting line in a box indicates this pointer to NIL.

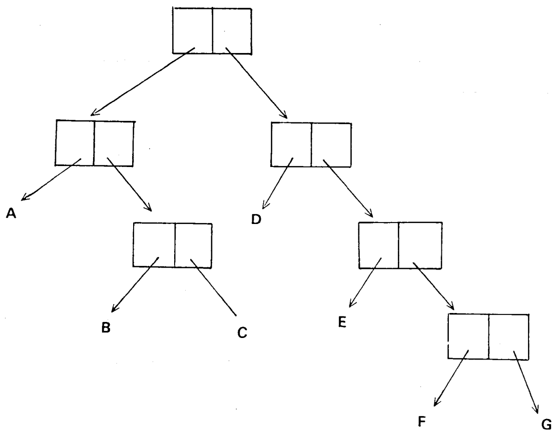


2.6 A list-structure is not unique. Even if the LISP read-routine reads the same lists, different, but isomorphous, structures will be created.

2.7 Dotted-pair. There is a notation in LISP called the dotted-pair notation. With this we can see an analogy with binary trees. A binary tree is a tree structure where every non-terminal node has two branches. We can follow the example



This binary tree corresponds to the following list structure.

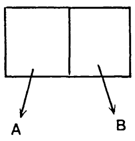


In dot-notation this is written as

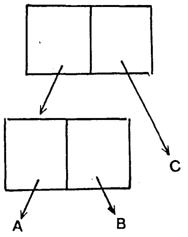
((A . (B . C)) . (D . (E . (F . G))))

The rule is that every node separates the tree in two parts, a left subtree and a right subtree. The dot is used to separate the two trees.

(A . B) corresponds to



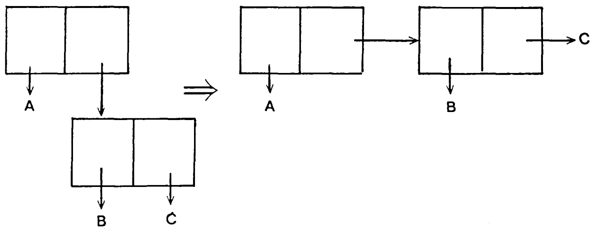
((A . B) . C) corresponds to



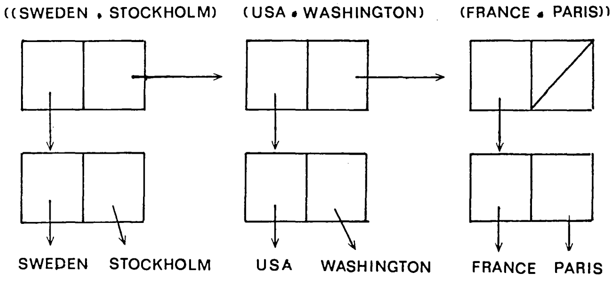
This dot-notation can be transformed to list-notation by following these rules:

- When a dot precedes NIL, the dot and NIL can be removed, (A . NIL) is identical to (A).
- When a dot precedes a left parenthesis the dot and the parentheses pair can be removed. (A . (B . C)) is identical to (A B . C)

This is easier to understand after looking at the following graphical notation



The use of the dotted-pair concept is rare because we usually see our structure as list-structure and not as binary trees. One example where it is used is the association list, which is a list of dotted-pairs.



When a dot is used in the meaning of a dotted-pair, it can only appear in the position before the last element. When a dot appears elsewhere, it is interpreted as an ordinary atom.

eg (A . B .) is a list with four elements

2.8 For the LISP read-routine it does not matter what notation we use when expressing a list. However, the LISP print-routines use the list-notation as much as possible and use the dot only when the second field in a cons cell points to a non-list, eg an atom.

The list (A B C) can be written as

(A B C . NIL)

(A B . (C . NIL))

(A . (B . (C . NIL)))

and all will create the same structure. The LISP print-routine will print (A B C).

(A . (B . (C . D))) will be printed as (A B C . D)

2.9 An <u>S-expression</u> can now be defined recursively as

a. a literal atom, number or string, eg A, 3 or "XY"

b. a dotted-pair of S-expressions, eg ((A . 12) . B)

<u>Exercises</u>

1. Write the structure for

a. (A B (C (D) E))

b. (((A) B) C)

c. (A <(B C D) E F> G>

d. (A (B . C))

e. (A (C D . E) (G . NIL) . (H I>

f. (. . . .)

g. ((A . B) (C . D) (E . F))

h. (A . (B . (C . NIL)))

i. (((A . B) . C) . NIL)

2. Which of the above expressions will be printed by the LISP print-routine but in a manner different from that read by the read-routine? How will they be printed?

3. Primitive functions

3.1 There are a great number of standard functions in a LISP system. These are already defined and exist in the system when entering. Our own functions can be introduced but this will be discussed at more length in Section 7. In the INTERLISP/360-370 there are about 400 functions. A user of LISP does not need to know all of these functions, but he must learn a number of them and know where in the manual to look for the remainder. This paper will cover the most important functions, either by giving the function definition or by giving a reference to the LISP manual where an index of all functions is to be found.

In this section we will introduce the functions car, cdr, (and their extensions), cons, equal, eq, atom, null and memb.

3.2 When describing LISP functions it is convenient to use the M-notation.[1] A functional-expression looks like

```
fn[arg1, arg2, ... , argn]
```

More about the M-notation is introduced later in the text.

3.3 Let us start by introducing some of the more elementary LISP functions. Notice that a function can behave differently according to the argument's datatype.

[1] There are two notations used in LISP. The first, introduced here, is M-notation (Meta-notation), which is quite similar to Algol-notation. The other, S-notation (S-expression notation), is introduced later.

In M-notation we distinguish very carefully between "program" and "data". Program, (such as functions, variables, if-then-else etc), is written in lower-case, while data, such as atoms, lists etc is written in capitals.

The S-notation is used, when running LISP, because both programs and data must then be expressed as S-expressions and there is no real syntactic difference between a program and data, and this introduces some problems.

In this paper we will use the M-notation during the introductory sections, and in later sections it can be used to describe forms such as

```
car[x]  and  memb[FOO,car[x]]
```

car[l] If *l* is a list, gives as value the first element of the list (called the *head* of the list).

eg car[(A B C D)] = A

car[((A B) C D)] = (A B)

car[NIL] is always NIL

l is an atom. Gives as value the atom's global value (see Section 15).

cdr[l] If *l* is a list, gives as value the list without the first element (called the *tail* of the list).

eg cdr[(A B C D)] = (B C D)

cdr[(A)] = NIL

cdr[(A . B)] = B

cdr[NIL] is always NIL

If *l* is atom, gives as value the atom's property-list (see 5.4).

3.4 By combining these functions we can find, for example, the second element on a list's fourth sublist, as in

(A (B) (C) (C (D E) F))

by car[cdr[car[cdr[cdr[cdr[(A (B) (C) (C (D E) F))]]]]]], which is (D E)

There are already functions which perform this kind of combination of *car* and *cdr*.

c*aa*r[l]	is identical to	c*a*r[c*a*r[l]]	
c*ad*r[l]	is identical to	c*a*r[c*d*r[l]]	
c*da*r[l]	is identical to	c*d*r[c*a*r[l]]	etc

The system supports functions with up to four *a* and *d* in it. The above example could have been written thus

cadadr[cddr[(A (B) (C) (C (D E) F))]]

3.5 cons[x,l] When *l* is a list, *cons* will give as value the list where *x* is the head and *l* is the tail. When *l* is an atom it returns a dotted-pair as described in 2.7.

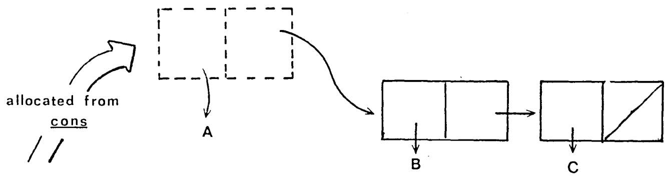
eg cons[A,(B C)] = (A B C)

cons[(A B), (A B)] = ((A B) A B)

cons[A, B] = (A . B)

cons allocates a new list-cell every time it executes.

cons[A, (B C)]



3.6 Some predicates will now be introduced. A predicate is a function giving a truth value, such as true or false. False is represented by NIL and true is represented by an arbitrary LISP element ≠ NIL. Normally one uses the atom T which has the initial value T[1].

equal[x,y] Tests if x and y are similar, in the sense that if x and y are of the same datatype, the LISP print-routine will print x and y identically. If they are similar it returns T, otherwise it returns NIL.

```
eg  equal[(A B), cdr[(X A B)]] = T
    equal[A, cadr [(X A B)]] = T
    equal["ABC", "ABC",] = T
    equal[A, car[((A))]] = NIL
```

Equal is normally used for comparing lists. There are more specialised functions, and therefore more efficient, for comparing different datatypes, such as

eq for literal atoms (see below).

eqp for numbers (see Section 12).

strequal for strings (see Section 22).

eq[x,y] Tests if x and y are identical, in the sense that the pointer are identical, If they are identical it gives the value T, otherwise NIL.

```
eg  eq[A, car[(A B)]] = T

    eq[A, cadr[(A B)]] = NIL
```

Eq is normally used for comparing atoms. Remember that atoms are unique. It follows that pointers to the same atom are identical.

[1] NIL and T are system variables and should not be used as variables by the user.

atom[x]	Tests if _x_ is an atom (literal atom or number) and returns T, otherwise NIL.

```
eg   atom[car[(A B)]] = T
     atom[12.34E4] = T
     atom[(A B)] = NIL
```

null[x]	Tests if _x_ is NIL (the empty list) and if so returns T, otherwise NIL.

```
eg  null[NIL] = T
    null[T] = NIL
    null[()] = T
```

memb[x,l]	Tests if _x_, normally an atom, is an element on the top level on the list _l_ and gives then a _true_ value (≠NIL), otherwise it gives the value NIL.

```
eg  memb[X,(A X B C)] = (X B C)
    memb[Q,(A (Q) B)] = NIL
    memb[Z,((X Y Z) Z)] = (Z)
    memb[3,(1 2 3 4)] = (3 4)
    memb[1.2, (2.1 1.2)] = NIL
```

The actual value returned from _memb_ is the rest of the list _l_, where _x_ is the first element.

Exercises

1. Combine functions for testing if the third element of (A B (X Y) C) is an atom.

2. Combine functions for testing if the first sublist's second element in ((A (Q Q) (A A)) (A A)) is similar to (Q Q).

3. Construct a list of the elements ADAM, (BERTIL) and ((CAESAR)).

4. What value is returned from these expressions.

   a. caddr[cadar[((A (B C D E)) F G)]]

   b. eq[A, cdar[((A B A))]]

   c. cons[cadar[((A (B (C D))) E),(Q Q)]]

   d. cons[(= =), =]

   e. caar[NIL]

   f. null[cddddr[(A B C)]]

   g. cadr[(A . (B . C))]

h. cddar[((A . (B . (C . D))) . E)]

i. memb[cadar[((+ ? -) /)], (: + / ? - *)]

5. Suppose l has the value (A ((B C) D))
Is it true that

a. equal[l, cons[car[l], cdr[l]]]

b. memb[C , caadr[l]]

c. equal[cons[cdr[l], cdadr[l]], (((B C) D) D)]

4. Conditional expression

4.1 A conditional expression in LISP is written in M-notation as Algol's if-then-else. Usually we have several tests and branches so it is convenient to introduce elseif. The expression has the following form

```
    if $p_1$ then $e_1$
elseif $p_2$ then $e_2$
elseif $p_3$ then $e_3$
        .
        .
        .
  else $e_{n+1}$
```

p_i and e_i can be arbitrary LISP-expressions including other conditional expressions. The evaluating rule for this is

- evaluate p_i in order until the first p_k, which has the value true (≠NIL). Then evaluate e_k, its value will be the value of the whole conditional expression.
- if all p_i are false then e_{n+1} will be evaluated and its value will be the value of the whole conditional expression.

4.2 Suppose we want to count the number of elements in a list and return that number if less than 3, otherwise return the value MANY. The conditional expression for this is

```
    if null[l] then 0
elseif null[cdr[l]] then 1
elseif null[cddr[l]] then 2
elseif null[cdddr[l]] then 3
  else  MANY
```

If l is (A B) the value will be 2.

4.3 There is an extension of this conditional expression described by the following example

```
      if p1 then e1
  elseif p2 then e21  e22  e23
  elseif p3
  elseif p4 then e4
```

The evaluation of p_i is the same, but if p_2 is true then all forms e_{21} to e_{23} will be evaluated in order and the value of the conditional expression is the value of e_{23}, the last form. This corresponds to Algol's begin ... end parenthesis, and in LISP it is called implicit progn. If p_3 is true (≠NIL) this value will be returned as the conditional expression's value. This is used instead of writing

elseif p_3 then p_3

in which case we must evaluate p_3 twice.

The general rule is that we can have arbitrary numbers of expressions after then or that then can be omitted completely.

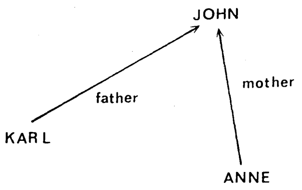
If there is no else statement and all pi are false then NIL will be returned as value.

4.4 Later we have examples which explain these conditional expressions in more detail so in this section we have omitted the exercises.

5. Property-lists

5.1 Every literal atom has an associated property-list. This section describes how to use them and introduces the functions put, getp, addprop and remprop, which are used for manipulating property-lists.

5.2 In this example we have used the property-list to store facts about family relationships.



KARL, ANNE and JOHN are objects; father and mother are relations. A fact can then be represented as an object - relation - object triple. LISP gives now a very convenient way to store these triples on property-lists.

We can store "KARL is FATHER of JOHN" and "ANNE is MOTHER of JOHN" and then retrieve "who is JOHN's FATHER? and "who is JOHN's MOTHER?"

This is stored by

```
put[JOHN,FATHER,KARL]
```

and we say that the carrier JOHN has under the property FATHER the property value (or shortly value) KARL.[1] The carrier and the property[2] must be literal atom and the property value can be of arbitrary type.

[1] These names can be confusing. In some LISP systems it is said that an atom under an indicator has a property.

[2] Actually the property can be of arbitrary type, but the normal property-list functions, such as put and getp make the comparision of the property by eq.

We can then store

    put[JOHN,MOTHER,ANNE]

To retrieve we do

    getp[JOHN,MOTHER] and get the value ANNE and
    getp[JOHN,FATHER] gives KARL

5.3 Here are some functions used for property-lists.

put[atm,prop,val]	Stores on atm's property-list under the property prop the value val. If there already was a value stored under that property it will be over-written by the new value. The value return from put is val. eg put[A,B,C] = C. On A's property-list C is stored under the property B.
getp[atm,prop]	Gets the value under the property prop on the atom atm's property-list. If there is no value NIL is returned. eg getp[A,B] = C, if we assume the above put. getp[A,X] = NIL
addprop[atm,prop,new]	Adds new to the value stored on atm's property list under the property prop. The value returned is the new value. eg addprop[X,Y,Z] = (Z). Under Y the list (Z) is stored on X's property-list. addprop[X,Y,W] = (Z W). W is added to the list. getp[X,Y] = (Z W)
remprop[atm,prop]	Removes on atm's property-list the value under the property prop. Also prop is removed. The value returned is atm. eg remprop[X,Y] = X getp[X,Y] = NIL

In the above functions atm, and prop must be literal atoms and val and new can be of arbitrary types.

5.4 Let us continue with the family relationships introduced at the beginning of this section and see how we can store and retrieve data on the property-lists.

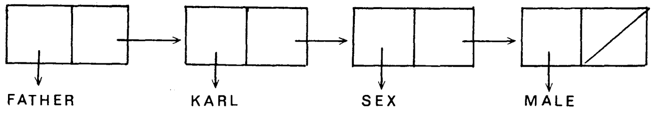
Suppose the following holds	Corresponding LISP expressions
Karl is father of John.	put[JOHN, FATHER, KARL]
Wilhelm is father of Karl.	put[KARL, FATHER, WILHELM]
Karl's children are John, Mary and Jim.	put[KARL, CHILDREN, (JOHN MARY JIM)]
John is a male.	put[JOHN, SEX, MALE]
Karl is married to Anne	put[KARL, MARRIED, ANNE]
Karl has one more child Tim.	addprop[KARL, CHILDREN, TIM]
Jim is of the same sex as John.	put[JIM, SEX, getp[JOHN, SEX]]
The one Karl is married to has the same children as Karl and one more child Eva.	put[getp[KARL, MARRIED], CHILDREN, cons[EVA, getp[KARL, CHILDREN]]

5.5 The implementation of property-lists makes it possible to retrieve an atom's property-list simply by doing <u>cdr</u> of the atom. The property-list can contain some system properties and we are not allowed to remove or change them in any way. Be careful about this!

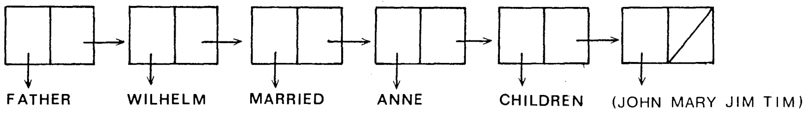
5.6 The property-list is actually an ordinary list, where every second element is a property and the other a value.

From the above examples we have the following structures

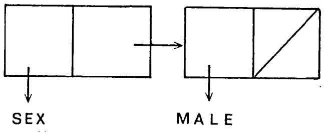
<u>John's property-list</u>



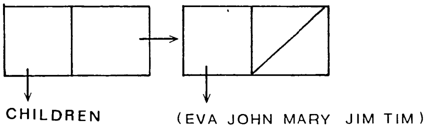
<u>Karl's property-list</u>



Jim's property-list



Anne's property-list



5.7 We require answers to the following questions — Corresponding LISP expressions

We require answers to the following questions	Corresponding LISP expressions
Give the name of one of Karl's children	car[getp[KARL, CHILDREN]]
Do we know Mary's sex?	getp[MARY, SEX]
Is Karl married to someone who has a child named Eva?	memb[EVA,getp[getp[KARL,MARRIED], CHILDREN]]
Is Jim a male?	eq[getp[JIM, SEX], MALE]
Is Anne married to Karl?	<u>if</u> eq[getp[ANNE, MARRIED], KARL] <u>else</u> eq[getp[KARL, MARRIED], ANNE]
Who is John's father's father?	<u>if</u> getp[JOHN, FATHER-FATHER] <u>elseif</u> getp[getp[JOHN, FATHER], FATHER]

Notice in the last conditional expression that there is no <u>then</u> and no <u>else</u> expression. This was described in Section 4.3.

Changes to the property-list	Corresponding LISP expressions
Anne and Karl are not married any longer	remprop[KARL, MARRIED]
Viktor is father of Karl	put[KARL, FATHER, VIKTOR]

5.8 The above examples illustrate how to process property-lists with the functions introduced so far. Of course there are other ways of processing, but as yet we have not gained sufficient knowledge. An example is,

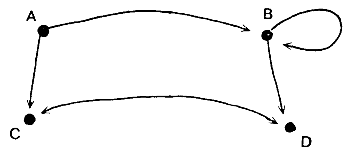
JOHN is no longer KARL's child.

With the function remove (see 11.6) this can be stated as

```
(PUT 'KARL 'CHILDREN (REMOVE 'JOHN (GETP 'KARL 'CHILDREN)))
```

Exercises

1. Suppose we wish to store a directed graph, such as



Decide the properties we need for storing the graph on property-lists. We must be able to answer questions such as

a. What nodes follow B?

b. What nodes precede C?

c. Does C follow A?

d. Can we go from A to D only through C?

e. Is the arc between C and D directed in both directions?

f. Is there a loop (an arc which starts and ends at the same node) at B?

6. S-notation

6.1 As yet nothing has been said about how to write a correct LISP expression suitable for use in the machine. The notation used for for this is called S-notation. This section will give the transformation rules for converting M-notation to S-notation.

6.2

	M-notation	S-notation
Variables	alpha	ALPHA

Variables in the meaning "its value", must be converted to corresponding atoms.

Functional expressions	$fn[arg_1, \ldots , arg_n]$	(FN ARG1 ... ARGN)

The expression is converted to a list where the first element is the function name and the remaining elements are the arguments.

Conditional expressions	if p_1 then e_1 e_{21}
	elseif p_2 then e_{22}
	elseif p_3
	else E_4
		(COND (P1 E1)
		(P2 E21 E22)
		(P3)
		(T E4))

The conditional expression is converted to a cond-expression. It consists of sublists, where every sublist corresponds to one of the test cases in the conditional expression. The first element of the sublist is the predicate and the rest are the expressions to evaluate. In the else branch T indicates that the following expressions will always be evaluated. T has always the true value.

Constants as	ADAM	'ADAM
atoms, lists	(L I S P)	'(L I S P)
and strings	"STRING"	' "STRING"

6.3 One of the main differences between LISP and the conventional programming languages, such as FORTRAN, COBOL, PL/1 etc, are that these languages have different representation of "programs" and "data", but in LISP there is no such difference. A "program" in LISP is simply represented as a list structure.

6.4 Quote. One difficulty is apparent, however, when using the same representation of "program" and "data". How can we separate them? The '-sign called quote-sign is introduced for this reason. Let us see some examples

M-notation	S-notation
car[(A B C)]	(CAR '(A B C))
eq[Q, car[(Q W)]]	(EQ 'Q (CAR '(Q W)))
cons[L, (I S P)]	(CONS 'L '(I S P))
null[l]	(NULL L)
memb[x, (A B C D E)]	(MEMB X '(A B C D E))
car[cdr[l]]	(CAR (CDR L))
car[(CDR L)]	(CAR '(CDR L))

In M-notation, variables (with lower case letters) are easily separated from constants (written in capital letters), but in S-notation the '-sign will tell that the following expression shall be interpreted as a constant. If a list is not '-ed (quoted) it is taken to be a functional expression, where the first element is the function name and the rest of the elements are arguments. See the difference in the last two examples above and really try to understand how necessary the '-sign is in the last example.

6.5 Cond is a special function which does not really follow the rule of quoting the arguments. These different function types are described in detail in Section 14.

6.6 The '-sign will be translated by the LISP read-routines to QUOTE, so that 'A becomes (QUOTE A) and '(A) becomes (QUOTE (A)).

Quote is a LISP-function, but works in such a way that it only returns its argument and by this no further evaluation will be performed.

The '-sign is only an abbreviation for QUOTE, so we can write it in both ways.

```
eg   '(A B 'C) - is similarly written  - (QUOTE (A B (QUOTE C)))
```

The LISP print-routines will print QUOTE instead of ' .

6.7 Numbers, NIL and T do not need to be quoted. They are so defined that they have themselves as value.

```
eg   cons[3, cons[T, cons[A, cons[(B), cons["S", NIL]]]]]
     (CONS 3 (CONS T (CONS 'A (CONS '(B) (CONS ' "S" NIL>
```

Exercises

1. Translate the following expressions from M-notation to S-notation:

```
a.  cdr[(A B C)]

b.  equal[A, car[((A))]]

c.  atom[12.34E4]

d.  equal[l, cons[car[l], cdr[l]]]]

e.  memb[C, caadr[l]]

f.  put[getp[KARL, MARRIED], CHILDREN
        cons[EVA, getp[KARL, CHILDREN]]]

g.  if null[l] then NIL else cdr[l]

h.  if eq[getp[ANNE, MARRIED], KARL] then T
       else eq[getp[KARL, MARRIED], ANNE]

i.  if getp[JOHN, FATHER-FATHER]
    elseif getp[getp[JOHN, FATHER], FATHER]
```

7. User-defined functions and assignment

7.1 User-defined functions. In LISP we can introduce our own functions. In S-notation we can write

```
(DE FOO (L) (CONS (CAR L) (CADDR L)))
```

We have now defined a function, named foo, with one argument l. The function is defined to make a dotted-pair of the first and third element on a list.

```
(FOO '(A B C D)) = (A . C)
(FOO '(A B (C D))) = (A C D)
```

7.2 The general form for function definition is

(DE fn (arg_1 arg_2 ... arg_n) $fnbody_1$ $fnbody_2$ ... $fnbody_m$)[1]

fn is the name of the function being defined.

arg_i must be a literal atom. Do not use the atoms NIL, T and NOBIND, they are used for other purposes. NIL and T represent the values false resp true and all atoms are initialised to the value NOBIND.

The number of arguments is arbitrary, (including no arguments).

$fnbody_i$ must be a LISP-expression - anything which can be evaluated in LISP's sense. At function call the bodies will be evaluated in order, singularly, and the value of the last body will be the value of that function call.

Functions defined in this way are called eval-spread functions. More about the different types of functions is described in Section 14.

7.3 Examples

The following function definition

```
(DE MARRIED (X Y) (PUT X 'MARRIED Y) (PUT Y 'MARRIED X))
```

[1] In M-notation a convenient way to write this general form can be

fn[arg_1, arg_2, ... , arg_n] == $fnbody_1$, $fnbody_2$, ... , $fnbody_m$

is defined by two function bodies. It will store that x has the value y under the property MARRIED and that y has the value x under the same property. The value of this function is the value of the last put, and its value is x.

```
(MARRIED 'ADAM 'EVA)
```

will perform the side-effects to store on ADAM's and EVA's property-lists and return the value ADAM. This demonstrates that we are not always interested in a function's value, but only in its side-effects. If we will have the value OK we could define it as

```
(DE MARRIED (X Y) (PUT X 'MARRIED Y) (PUT Y 'MARRIED X) 'OK)
```

Define a function before which checks if an atom x precedes the atom y on the list l, and then returns the value T otherwise the value NIL.

```
eg  (BEFORE 'C 'E '(A B C D E F G H)) = T
    (BEFORE 'T 'R '(Q R S T U V)) = NIL
```

The definition of before is[1]

```
(DE BEFORE (X Y L) (COND ((MEMB Y (MEMB X L)) T)
                         (T NIL)))
```

Note the use of the value of memb (see Section 3.6). In the first example above, the evaluating order in before will be

```
a.  (MEMB 'C '(A B C D E F G H)) = (C D E F G H)
b.  (MEMB 'E '(C D E F G H)) = (E F G H)
c.  (COND ('(E F G H) T) (T NIL))  will of course be evaluated to T.
```

7.4 Assignment. As in other programming languages we can assign values to variables. In LISP every literal atom can be a variable.

When we write

```
(SET 'L 'A)
```

we mean that the atom L is treated as a variable and gets as value the atom A. If we then write

```
(SET L 'Q)
```

we mean the atom L is a variable and the value of that atom, the atom A, gets the value Q. This is very important in LISP. In some other languages an identifier is always treated as a variable and

[1] If we can accept other true values than T we could also define before as

```
(DE BEFORE (X Y L) (MEMB Y (MEMB X L)))
```

when we write the variable we mean either its value as in

A + B + C + 4 is 15 if the variables A = 2, B = 4 and C = 5

or the variable itself as in

A = 10

It is the position of the variable in the expression which determines its interpretation.

In LISP however, a variable can have another variable as value and we must therefore distinguish very carefully in every situation if we mean the atom itself or its associated value. Further examples clarify this

(SET 'L '(A B))	L is assigned the value (A B)
(SET 'A 10)	A is assigned the value 10
(SET 'B 'X)	B is assigned teh value X
(SET B 'Y)	X (the value of B) is assigned the value Y
(SET X A)	Y (the value of X) is assigned the value of A which is 10

Now the following values exist

A	10
B	X
L	(A B)
X	Y
Y	10

7.5 The first argument to <u>set</u> can be an arbitrary LISP-expression, which evaluates to an atom.

eg (SET (CAR '(A B C)) 10) assigns the value 10 to A.

7.6 Normally when assignments are done the first argument in <u>set</u> is <u>quote</u>-ed. There is a special function <u>setq</u>, which makes this <u>quote</u>-ing implicit and which is more commonly used than <u>set</u>.

eg (SETQ A 'VALUE) is identical to (SET 'A 'VALUE)

7.7 We will not discuss the scope of a variable in LISP in this section as this is dealt with in Section 15. Until then we normally use the assignment for giving a <u>global value</u> to an atom.

as
use

Exercises

1. Define a function cd5r, which gives the fifth element on a list.

2. Define the following functions, which work in the family relationship example in Section 5.

   a. A function married[x,y] which checks if _x_ and _y_ are married. The function must look on both _x_'s and _y_'s property-lists. Return the value YES if they are married and NO otherwise.

   b. A function son[x,y] - meaning _x_ is son of _y_ - and which on _x_'s property-list stores that _y_ is father of _x_ and on _y_'s property-list adds that _x_ is son of _y_.
   Return the value OK.

   c. A function fatherofq[x,y] which checks if _x_ is father of _y_. Return YES or NO.

3. Suppose we do the following assignments

```
(SETQ R '(A B C))
(SET 'L 'R)
(SETQ X L)
(SET L (CAR R))
(SET R '(Q R S))
(SET (CAR A) (CDR A))
```

Which of the following expressions are true (≠NIL)

a.	(ATOM R)	f.	(EQ X 'R)
b.	(ATOM A)	h.	(CDDR A)
c.	(ATOM X)	i.	(CDDR Q)
d.	(EQ X L)	j.	(EQUAL Q (CDR A))
e.	(EQ X 'L)	k.	(EQUAL L (CADR A))

8. Running LISP

8.1 The LISP interpreter

A LISP system is interpretative. This means that the system reads a LISP-expression, and then directly calls an interpreter, which evaluates this expression, and writes as output the value of that evaluation. Its contrast is a compiled system, similar to FORTRAN, COBOL etc, where the expressions (programs) will be translated to machine code by a compiler. This machine code can then later be executed.

There is a LISP compiler which can be used for compiling LISP functions from the list structure format to machine code. The reason for compiling a LISP function is for efficiency but is not necessary for running LISP. The compiler will be described in Section 29.

8.2 The INTERLISP/360-370 can be used both as an interactive system and as a batch system. We assume that we first use the interactive system and see how it functions. Later we will see what changes to make and the differences when using it as a batch system.

First we must enter the LISP system. The exact procedure for this is machine and implementation-dependent and we must consult a local guide for this. When the system is entered it prompts us with a character which says that it is ready for input. The prompt character is — (be sure that your installation has not changed it). There are other prompt characters to be used when in other modes, such as in break and edit mode.

The LISP system works in the following loop:

- print a prompt character
- read a LISP expression
- evaluate this expression
- print its value

A conversation can appear thus

```
-(CONS 'A '(B C))↵                  Prompt character; give input and
                                    end the input by return and here
(A B C)                             is the value.

-(DE FOO (L) (MEMB V L>↵            Notice that all LISP expressions
                                    have a value. In some cases this
FOO                                 value is of no real interest, as
                                    in these two expressions.
-(SETQ V 'LISP)↵

LISP

-V↵

LISP

-(FOO '(ALGOL LISP FORTRAN))↵

(LISP FORTRAN)

-'A↵

A

-(QUOTE A)↵

A

-(EXIT)↵                            The function exit returns from
                                    LISP to the time-sharing monitor.
```

8.3 The LISP read-routine reads one LISP expression (atom, list etc) in S-notation. The form of writing it is free and the expression can be written on several lines (cards). The read-routine continues to read until it has read in a full expression, ie a full parenthesised list. Blank is ignored and used only for separating atoms, strings etc. However, it is important to find a good "style" to write LISP, so it can be read by other programmers. Normally LISP programmers follow these rules:

- one blank between atoms and between an atom and a left parenthesis
- no blank after a left parenthesis and no blank before a right parenthesis
- one blank between a right parenthesis and an atom

- one blank between right and left parenthesis
- same parenthesis following each other have no blanks between.

```
(A B (C D) (X (Y Z)) W)
```

is preferable to

```
(A B(C D ) ( X(Y Z))W )
```

When we have long lists we try to split them up on several lines (cards) and make the necessary indentations to describe the structure of the list. A cond-expression can appear

```
(COND ((NULL L) (FOO L))
      ((MEMB 'X (CDR L)) (FIE (CONS (CAR L) (CDDR L)))
                         (FIE (CONS (CADR L) (CDR L))))
      (T (FOO (CDR L))))
```

8.4 The input must of course be something which can be evaluated. A LISP expression given to the evaluator is called a form. If a form is a

- literal atom, i. e. used as a variable, is evaluated to its value. If it has no value an error occurs and the message U.B.A (UnBound Atom) indicates this.
- number , is evaluated to itself

eg

```
-(SETQ VAR 'VALUE)      Input
VALUE                   Value
-VAR
VALUE
-WAR                    The variable war has no value.
U.B.A                   The error message for an unbound
WAR                     atom is printed together with the
                        variable.
-123
123
```

- list, the first element is taken as a function and the remaining elements are arguments to that function. If the first element is not a function the system prints the error U.D.F (UnDefined Function).

eg

-(CAR '(A B C>	Input
A	Value
-(KAR '(A B C>	
U.D.F	This message indicates that kar is the undefined function.
KAR	
-(CAR (A B C>	Note the importance of the quote sign. The list (A B C) is now treated as a form and the system looks for a function A, which is not defined.
U.D.F	
A	
-	

If the error occurs deep in the function (many function calls have been made before the error is made) the system goes in a break. It prompts us then with a : . It gives us the possibility to analyze in detail the error and correct it and then go on with the evaluation. This will be described later; what we can do in such situations is to write | or (RESET) and return. We return to the LISP system's top level.

8.5 When we have parenthesis error the message can be confusing.

eg

-(CAR 'A B C>	We meant (CAR '(A B C>
U.B.A	The arguments to car are evaluated and its second argument B has no value and an U.B.A error message is printed.
B	
-(EQ (CAR X) CAR Y>	We meant (EQ (CAR X) (CAR Y))
U.B.A	When a function is an unbound atom it is nearly always parenthesis error or
CAR	
-(COND ((NULL L) NIL) - (CDR L>	provided we have not forgotten the T in the last clause in a cond-expression. In this case CDR is the predicate to evaluate and the error indicates that it has no value.
U.B.A	
CDR	

```
-(CADR '((A . ) (B . 2)) (C . 3>
U.D.F

C
```

The additional left parenthesis after the pair (B . 2) will cause (C . 3) to be a form for evaluation and C is then taken as a function.

```
-(DE FOO (X (CAR X))
-
-
-
-
```

We thought the expression was correct but the missing parenthesis after the first X will make the system prompt us to continue the expression. This can be very confusing because the system prompts for more and more input whatever is wanted. In this case a number of right brackets (>) can complete the expression.

8.6 We have now arrived at the stage where we can sit down and practice on the system. However, first a few more rules are necessary. When introducing a new function it is not permissable to use a function name already existing in the system. It is of course very difficult to know all the names. The system will print when we redefine a function

```
(CAR REDEFINED)
```

When this message comes, we must do

```
(MOVD 'CAR 'KAR)
(UNSAVEDEF 'CAR)
```

which re-stores the ordinary function definition, and gives the new definition the function name kar.

If we have defined a function foo we can look at it by doing

```
(PP:: FOO)
```

The function will be prettyprinted. Do this and see what happens.

When typing to the terminal it is important to know

- how to input a line to the computer
- how to delete characters from a line.
- how to make an interrupt and to come to LISP's top level again. This is done if our function goes in an infinite loop or if the print-routine prints a circular list and we want to stop it.[1]
- what character set we can use. Is it permissable to have small letters and other available characters. For this consult the local LISP guide for that terminal.

[1] An attention-D causes return to the top-level; an attention-PO stops the printing of an expression, see further 30.5.

8.7 Running LISP in batch. This can be done either by a remote job terminal or by punching cards. We must find out which control statements (JCL statements for IBM) we require.[1] Following them we can give the LISP expressions we want to have evaluated.

```
eg  // EXEC ULISP }        control statements
    (CAR 'A B C>  |
    (SETQ A 10)   |
    A             }        LISP statements
    (EXIT)        |
    /*
```

The output will then appear

```
    -----------------------------
    (CAR (QUOTE (A B C)))
    A
    -----------------------------
    (SETQ A 10)
    A
    -----------------------------
    A
    10
    -----------------------------
    (EXIT)
```

If an error occurs the system prints a message in the interactive way. If the error occurs deep, the system will print a backtrace. This backtrace contains information stored on the system's stacks. We will not discuss that information here but will return to it later in this paper.

8.8 There is a comment facility in INTERLISP/360-370. A form started by * is treated as a comment.

```
eg  (* THIS IS A COMMENT)
```

This expression is an ordinary form, but * is defined not to evaluate the list, but it returns a value. This means that a comment can only appear where the value does not effect the evaluation.

```
eg  (DE FOO (X) (* FOO IS A FUNCTION ...) (FIE X))


    (DE FOO (X) (COND ((EQ X 'A) (* THIS TESTS ...) (FIE X))
                      ((EQ X 'B) (* THIS ...) (FUM X))
                      (T (* THIS ...) (GUM X))))
```

are all OK, but not

```
    (DE FOO (X) (FIE X) (* FOO IS ...))
```

[1] In the Appendix is described the actual control statements needed to run.

Exercises

Sit down at a terminal and enter the LISP system.
Do the following:

a. Start by testing the standard functions introduced in Section 3. Test them with different types of arguments and really understand how they are working. Do not forget the put the '-sign in the right position. If errors such as U.B.A and U.D.F arise it is probable that you have forgotten a '-sign or that you have miss-spelled a function name. Do not give up until it is working correctly. The use of ' is very important and it is pointless to continue in this text if its use is not clear.

b. Make assignments. The '-sign is also here very important. Test exercise 3 in Section 7.

c. Define your own functions. Use conditional expressions in the definitions. Pretty-print the functions. If they are not working redefine them again and go on testing.

d. Find a problem where you can use the property-lists. One problem could be the family relationships introduced in Section 5. Introduce more properties and store facts on the property-lists. Define functions for making retrieval from the property-lists.

9. Recursive functions

9.1 For writing algorithms for symbolic datatypes such as lists etc, it is often desirable to define the algorithm in a recursive way. The use of recursive functions is very common in LISP and the language is designed to make recursion easy to handle. This means that in a function definition a function call to the function itself is allowed, either direct or indirect (a function a calls b which calls a).

Writing a recursive function is not a triviality for the beginner, especially if he is used to languages like FORTRAN, COBOL etc, and therefore writing programs in an iterative way, ie by using loops. Remember now that LISP is designed to be a functional language - we break our problem down to small functions, which call each other intensively - and that we for this reason cannot make programs in LISP by thinking in FORTRAN terms. To think in LISP we must learn and the only way to learn is to practice. This section will give many examples of recursive functions - then try to solve the exercises at the end of this section.

9.2 We start with the function ::memb,[1] described in Section 3.6.

```
(DE ::MEMB (X L) (COND ((NULL L) NIL)
                       ((EQ X (CAR L)) L)
                       (T (::MEMB X (CDR L)))))
```

The function ::memb is defined first to test if l is the empty list. It returns then the value NIL, which is obvious for no element can occur in an empty list. Then it tests if the first element on the list l is equal to x. If it is true, we have found an x, which is

[1] When definition of functions which already exist in the INTERLISP/360-370 system are given they are ::-marked. If a system function is redefined in an incorrect way it can break down the system, but of course its definition can be tested by using this ::-name.

an element of _l_ and therefore returns a _true_ value. Actually it is the list _l_ we return. If this was not the case we make a recursive call to _⁘memb_ again. But now we know that the first element of _l_ is not equal to _x_ and therefore we have the problem of testing if _x_ is an element of the list _l_, where the first element is removed. Whatever value that function call will return, _true_ (≠NIL) or _false_ (NIL) that value will of course also be the value returned from the function. This is obvious for the first element of _l_, does not effect the value for we know that this element was not equal to _x_.

The first two steps in _⁘memb_ are our _terminating_ criteria, which stops the recursion.

Let us clarify this discussion by following an example.

```
        ⁘memb[Q,(O P Q R)]
```

We get the following call structure

```
enter ⁘memb
X = Q
L = (O P Q R)
|
|   enter the conditional expression and evaluate
|   the last branch
|   |
|   |   enter ⁘memb
|   |   X = Q
|   |   L = (P Q R)
|   |   |
|   |   |   enter the conditional expression and
|   |   |   evaluate the last branch
|   |   |   |
|   |   |   |   enter ⁘memb
|   |   |   |   X = Q
|   |   |   |   L = (Q R)
|   |   |   |   |
|   |   |   |   |   enter the conditional expression
|   |   |   |   |   and evaluate the second branch. The
|   |   |   |   |   result from this evaluation is (Q R)
|   |   |   |   |
|   |   |   |   return from ⁘memb with value (Q R)
|   |   |   |
|   |   |   the conditional expression gives the value (Q R)
|   |   |
|   |   return from ⁘memb with value (Q R)
|   |
|   the conditional expression gives the value (Q R)
|
Return from ⁘memb with value (Q R), which is the final value.
```

9.3 We define a function ::remove[x,l], which returns as value a new list, where all occurences of the atom x are removed from the top level of the list l.

eg ::remove[A, (C A D A)] = (C D)

::remove can be defined

```
(DE ::REMOVE (X L) (COND ((NULL L) NIL)
                         ((EQ X (CAR L)) (::REMOVE X (CDR L)))
                         (T (CONS (CAR L) (::REMOVE X (CDR L))))))
```

The function starts to test if we are trying to remove anything from an empty list, which we of course cannot. This is our terminating criteria for ending the recursion.

The other two tests will go on in the recursion and remove x from cdr of l. The differences between the two tests are that we in the third test case put the first element - it can not be equal to x - on the list we get as value from the recursive call - this list contains only elements not equal to x, for they are now removed.

The above example will give the following enter/return structure. The entering and leaving of the conditional expression is not shown.

```
::remove:
X = A
L = (C A D A)

    ::remove:
    X = A
    L = (A D A)

        ::remove:
        X = A
        L = (D A)

            ::remove:
            X = A
            L = (A)

                 ::remove:
                 X = A
                 L = NIL

                 ::remove = NIL

            ::remove = NIL

        ::remove = (D)

    ::remove = (D)

::remove = (C D)
```

The listing of call/return structure of ∗remove, corresponds nearly to a trace-function, which exists in the LISP system. A traced function will at entry write the arguments and its associated values and at return write the computed value. For getting this trace we do

```
(TRACE ∗REMOVE ∗MEMB ...other functions we want to trace...)
```

To remove the trace we do

```
(UNBREAK ∗REMOVE ...other functions we want to untrace...)
```

9.4 Define a function ∗union[x,y], which takes two lists, where each list is supposed to be a list of atoms, and which makes a union of the two lists.

```
(DE ∗UNION (X Y) (COND ((NULL X) Y)
                       ((MEMB (CAR X) Y) (∗UNION (CDR X) Y))
                       (T (∗UNION (CDR X) (CONS (CAR X) Y)))))
```

A trace of ∗union[(A B C), (X B Y)]

```
∗union:
X = (A B C)
Y = (X B Y)

    ∗union:
    X = (B C)
    Y = (A X B Y)

        ∗union:
        X = (C)
        Y = (A X B Y)

            ∗union:
            X = NIL
            Y = (C A X B Y)

            ∗union = (C A X B Y)

        ∗union = (C A X B Y)

    ∗union: = (C A X B Y)

∗union = (C A X B Y)
```

In this function we use y for building the value and when all elements of x were taken (x = NIL) y was returned as value, and this value will be returned up as the final value.

9.3 Define a function totremove[x,l], which returns a value, where all occurences of x regardless of the level, are removed from l.

eg totremove[X,(A X (B (X C)))] = (A (B (C)))

```
(DE TOTREMOVE (X L)
    (COND ((ATOM L) L)
          ((EQ X (CAR L)) (TOTREMOVE X (CDR L)))
          (T (CONS (TOTREMOVE X (CAR L)) (TOTREMOVE X
                                            (CDR L))))))
```

In the cons we make two recursive calls and we say that we are making recursive calls both in the car- and cdr-direction. This is termed double recursion.

In the function we test if x is equal (with eq) to car[l], and if so we go on as in the function *remove. However, if they were not equal, l was either an atom (we assume no strings in the lists) not equal to x or a list. In that case we make a recursive call to totremove with that first element as l. The test atom [l] will find out if l is an atom and return then the atom as value. This test also takes care of the end of the list test, when l is NIL, because atom[NIL] is T. If l was a list, go on. The value of the first recursive call will be cons-ed on the list we get as value after the second recursive call.

Let us follow a trace of <u>totremove</u>:

```
totremove:
X = X
L = (A X (B (X C)))

    totremove:
    X = X
    L = A
    totremove = A

    totremove:
    X = X
    L = (X (B (X C)))

        totremove:
        X = X
        L = ((B (X C)))

            totremove:
            X = X
            L = (B (X C))

                totremove:
                X = X
                L = B
                totremove = B

                totremove:
                X = X
                L = ((X C))

                    totremove:
                    X = X
                    L = (X C)

                        totremove:
                        X = X
                        L = (C)

                            totremove:
                            X = X
                            L = C
                            totremove = C

                            totremove:
                            X = X
                            L = NIL
                            totremove = NIL

                          totremove = (C)

                    totremove = (C)

                    totremove:
                    X = X
                    L = NIL
                    totremove = NIL

                totremove = ((C))

            totremove = (B (C))

            totremove:
            X = X
            L = NIL
            totremove = NIL

        totremove = ((B (C)))

    totremove = ((B (C)))

totremove = (A (B (C)))
```

9.6 In the function totremove, we made the test atom [l]. This works correctly only in cases where the elements are atoms. If strings and other datatypes can be elements the test should be done by the function nlistp, which is true when its argument is not a list. NIL is here treated as an atom, so nlistp[NIL] = T.

```
eg   nlistp[ATOM] = T
     nlistp[(A T O M)] = NIL
     nlistp["STRING"] = T
     nlistp[()] = T
```

9.7 The function nlistp is also useful when using lists ending with a dotted-pair.

eg   The function ::copy[l], which makes a copy of a list structure, including dotted-pairs.

```
::copy[(A B (C . D) . E)] = (A B (C . D) . E)

(DE ::COPY (L)
(COND ((NLISTP L) L)
      (T (CONS (::COPY (CAR L)) (::COPY (CDR L))))))
```

If we suspect that a list can end with a dotted-pair a good rule is to use nlistp as the terminating critera, when processing lists, instead of null. This example shows what can happen if we have the null test as terminating criteria for a list but the list happened to end with a dotted-pair.

eg   take the function ::remove from Section 9.3 and evaluate
::remove[X,(A X . B)]
The trace gives,

```
::remove:
X = A
L = (A X . B)

    ::remove:
    X = A
    L = (X . B)

        ::remove
        X = A
        L = B
```

Here l has the value of atom B, and we go down and make the recursive call

::remove[A,cdr[B]]

What is cdr[B]? Cdr of an atom contains the property-list. If B's property list contains property/value pairs the function ::remove will go on there. The test with nlistp, would stop and prevent the function from doing unexpected things like this.

Exercises

All these examples can be tested on the computer. Use trace. Many of the functions mentioned here are functions which already exist in the LISP system, (they are ⁑-marked).

1. Define a function even[l], where l is a list, which gives as value T if there was an even number of elements on the list, otherwise NIL

   eg even[(A B C D)] = T

   even[((A) (B C) (D))] = NIL

2. Define a function append2[x,y], where x and y are lists, which gives as value the concatenated list of x and y.

   eg append2[(A B C D), (X Y Z)] = (A B C D X Y Z)

3. Define a function ⁑intersection[x,y], where x and y are lists treated as sets which gives as value the list corresponding to the intersection of x and y.

   eg ⁑intersection[(A B C D E), (Q D E W A Z)] = (A D E)

   The order of the elements in the value-list can differ, depending on the algorithm used.

4. Define a function ⁑reverse[l], where l is a list, which gives as value a list where the elements on top level are in reversed order. Hint - use append2 from exercise 2.

   eg ⁑reverse[(A B (Q W) C D)] = (D C (Q W) B A)

5. Define a function ⁑subst[x,y,l] where x and y can be of arbitrary type and l a list, which gives as value a new list in which every occurence of y on top level is substituted by x.

   eg ⁑subst[NEW, OLD, (A OLD (B OLD) OLD C)] =
   (A NEW (B OLD) NEW C)

6. Define a function totreverse[l], where l is a list, which gives as value the list where all elements are reversed on all levels

   eg totreverse[(A (B C) D ((E) F))] = ((F (E)) D (C B) A)

7. Define a function totsubst[x,y,l], which works as subst, but substitutes on all levels.

   eg totsubst[NEW,OLD,(A OLD (B OLD) OLD C)] =
   (A NEW (B NEW) NEW C)

8. Define a function ::sublis[al,l], where al is a list of dotted-pairs and l a list. For every pair in al the first element shall be substituted by the second element in the list l. The substitutions shall be done on all levels.

```
eg  ::sublis[((A . X) (B . Y)), (C (A B (D B)) A)] =
                                  (C (X Y (D Y)) X)
```

9. Define a function ::pair[x,y], where x and y are lists of the same length (they contain the same number of elements on top level). The value returned from pair, should be a list of dotted-pairs. The first element from x and y builds the first dotted-pair, the second element from x and y the second pair etc.

```
eg  pair[(ONE TWO THREE FOUR FIVE), (1 2 3 4 5)] =
      ((ONE . 1) (TWO . 2) (THREE . 3) (FOUR . 4) (FIVE . 5))
```

10. Define a function flatten[l], where l is a list, which builds a list where all parentheses - except the outer pair - are removed. The atoms in l shall come in the same order in the new list.

```
eg  flatten[((A B (C)) D (E (F (G)) H))] =
                              (A B C D E F G H)
```

11. Define your own sort-package by using a tree-sort algorithm. A function tree-sort[l], where l is a list of atoms should be obtained. Introduce a global variable PRECEDENCE with a value giving the ordering relation between the atoms.

```
eg  PRECEDENCE := (A B C D E F G H I J)

    treesort[(B A E G C I J G)] = (A B C D E G G I J)
```

The package should also contain a function merge, which makes a merge of two sorted lists.

```
    merge[(A C E F F I), (B D E G H J)] =
                (A B C D E E F F G H I J)
```

Hints - Define a function order[x,y], which is true if x precedes y on PRECEDENCE, otherwise NIL

```
    order[B,H] = T, or another true value
    order[H,F] = NIL
```

Define a function buildtree[l], where l is a list to treesort, which builds a sort tree. An example illustrates the tree-sort.

(D B E J C A)

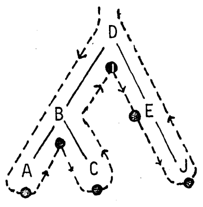
```
D  D       D        D         D
   /      / \      / \       / \
   B     B   E    B   E     B   E
                       \     \   \
                        J     C   J


     D
    / \
   B   E
  / \   \
 A   C   J
```

A new element enters the tree at the root, and at every node a comparison is made, by order, and if the new element precedes the node element, go to the left, otherwise go to the right. When coming to a terminal node, the new element inserts either to the left or to the right depending on the comparison.

Define a list structure corresponding to this tree.

Define a function walktree[l], where l is the sort tree, which walks through it by postorder traversal.[1]



The algorithm used at every node can be defined recursively by

- traverse the left subtree
- visit the node
- traverse the right subtree.

[1] Further information about binary trees can be found in Knuth's Fundamental Algorithms (ref 6).

If you follow this structure of programming that treesort is defined as

```
      (DE TREESORT (L) (WALKTREE (BUILDTREE L)))
```

There is a sort function in INTERLISP and it is shortly described in Section 30.

10. Introduction to break and edit

10.1 The INTERLISP/360-370 is designed to be used as a hard interactive system and much effort has been put in debugging packages. When an error occurs - ie a variable has no value (U.B.A), the system goes in a break. This now gives the user possibility to interact with the system, to be able to find out where the error occured and to correct the error, and then leave the break and continue the evaluation. A structure editor may be used to correct the error.

In this section some commands to break and edit are explained. More about break and edit and error-handling in general is to be found in Sections 24-26. At the end of this section there is a discussion that can be useful in a batch environment.

10.2 An example illustrates the ideas

```
-(DE REV (L RES) (COND ((NULL L) X) (T (REV (CDR L) (CONS (KAR L) RES>

REV

-(REV '(A B C))              Test rev by an example.

U.D.F                        An error has occured during evaluation.
                             There is an undefined function (U.D.F)
(KAR BROKEN)                 kar.

                             The break prompts us with : .
:L                           To break we can give either commands or
                             forms which will be evaluated. L is no
(A B C)                      command and is therefore treated as a
                             form and is evaluated.
:RES                         RES will also be evaluated.

NIL

:BT                          A command which prints the function calls
                             done before the error occured.
COND

REV

EVAL

LISPX
```

```
:(EDITF REV)                       Enter the editor and edit rev. The
                                   editor prompts with :: .
EDIT

::P                                Prints rev with printlevel = 2

(LAMBDA (L RES) (COND & &))

::3 P                              Look at the third sublist and print it.

(COND (& X) (T &))

::3 2 P

(REV (CDR L) (CONS & RES))

::3 2 P

(KAR L)                            Here is the error, car is mis-spelled.

::(1 CAR)                          Change the first element on the list
                                   to car.
::P

(CAR L)                            The error is corrected.

::OK                               Leave the editor.

REV                                In the break again.

:RETURN (CAR L)                    Give the broken expression, the value
                                   of (CAR L), which was calculated to A.
KAR = A

U.B.A                              An Error again, X has no value, an
                                   unbound atom (U.B.A)
(X BROKEN)

:(EDITF REV)

REV

::(R X RES)                        Replace all X by RES.

::OK                               Leave editor

REV

:RETURN RES                        Leave the break and return the value of
                                   RES. The evaluation continues and the
(C B A)                            reversed list is printed.
```

```
-(PP REV)                    Back to LISP's top level again.

(REV
   <LAMBDA (L RES)
       (COND
          ((NULL L)
             RES)
          (T (RES (CDR L)
                 (CONS (CAR L)
                       RES>)
(REV)
```

The break gives us the possibility to search for where the error occured. There are commands by which we can see the chain of function calls done (BT above), the actual value of variables and we can evaluate arbitrary forms so we are allowed to do whatever computation we will at this break point.

The editor gives us then possibility to correct errors or any changes in the code we want to perform.

10.3 Break commands[1]

STOP or |	Leaves the break to either a higher break or to LISP's top level.
BT	Prints a backtrace of all functions which were called before the error.
BTV	Prints the functions, variables and values.
RETURN form	Returns from the break and the value of form is given as value to the expression which caused the break.
form	If form is not interpreted as a break command it is evaluated and the value is printed.
?	Writes a list of available break commands.

10.4 Edit commands

To invoke the editor there are three functions, editf (for functions), editv (for global values) and editp (for property-lists).

[1] A break can be initiated by an attention-B or attention-H, see further Section 30.

At every moment the editor's attention is centered to a substructure of the expression being edited. It is called the current expression, cexp. A faulty command is responded by ? and cexp is then not changed.

P	Prints the cexp, with print level = 2
?	Prints the cexp on all levels
PP	Prettyprints cexp
n	An integer, if positive set cexp to the nth element of the current cexp and if negative the nth element from the end. If n=0, cexp will be the last cexp before.
|	Sets cexp to the top level expression again.

eg

```
-(DE FOO (X) (AND X (FIE (CAR X) (CADR X))))

FOO

-(EDITF FOO)

EDIT

::P

(LAMBDA (X) (AND X &))

::3 ?

(AND X (FIE(CAR X) (CADR X))))

 ::P

(AND X (FIE & &))

::0 P

(LAMBDA (X) (AND X &))

::OK

FOO

-
```

(n)	n⩾1 deletes the nth expression of cexp.
(n e_1 ... e_m	n⩾1 replaces the nth expression by e_1 to e_m.
(-n e_1 ... e_m)	n⩾1 inserts e_1 to e_m before the nth element.
(N e_1 ... e_m)	adds e_1 to e_m to the end of cexp.

```
eg  -(EDITF FOO)
    EDIT
    ::(2 (Y)) P
    (LAMBDA (Y) (AND X &))
    ::3 (2) P
    (AND (FIE & &)
    ::(-2 Y (FUM (CAR Y))) P
    (AND Y (FUM &) (FIE & &))
    ::(N CAR) 0 ?
    (LAMBDA (Y) (AND Y (FUM (CAR Y)) (FIE CAR X)
    (CADR X)) CAR))
    ::OK
    FOO
    -
```

F expr	If _expr_ is an atom it searches for _expr_ in _cexp_, such that car[cexp] = expr. It searches first on top level of _cexp_ and then from the beginning of _cexp_ on all levels. If _expr_ is a list it searches the first occurence in _cexp_ regardless of levels. _cexp_ is then equal to searched expression
(R x y)	All occurences of _x_ are replaced by _y_.
UP	Sets _cexp_, so that _car_ of new _cexp_ is equal to old _cexp_.

```
eg  -(EDITF FOO)
    EDIT
    ::F FIE P
    (FIE (CAR X) (CADR X))
    ::(R X Y) P
    (FIE (CAR Y) (CADR Y))
    ::UP ?
```

```
... (FIE (CAR Y) (CADR Y)) CAR)

*F CAR P

... CAR)

*(1 END) 0 0 P

(AND Y (FUM &) (FIE & &) END)

*OK

FOO

-
```

(BI n m)	both in. A left parenthesis is inserted before the nth element and a right parenthesis is inserted after the mth element.
(BI n)	as (BI n n)
(BO n)	both out. Removes both parenthesis from the nth element.

eg If cexp is (A (B C) D E)
(BI 2 3) gives (A ((B C) D) E)
(BI 3) gives (A (B C) (D) E)
(BO 2) gives (A B C D E)

(LI n)	left in. Inserts a left parenthesis before the nth element and a corresponding right parenthesis at the end.
(LO n)	left out. Removes the left parenthesis from the nth element. All elements after the nth element are deleted.
(RI n m)	right in. Inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression.
(RO n)	right out. Removes the right parenthesis from the nth element, moving it to the end of the current expression. All elements following the nth element are moved inside the nth element.

```
eg  cexp is (A (B C) D E)
    (L1 3) gives (A (B C) (D E))
    (LO 2) gives (A B C)
    (RI 2 1) gives (A (B) C D E)
    (RO 2) gives (A (B C D E))
```

UNDO	Undo-es the last change done. It is used if for instance a wrong element was deleted. This gives the user the ability to experiment with the editing commands. All of the commands are undoable.
|UNDO	Undo-es all changes done.
OK	Leaves the editor.

10.5 In batch some of these facilities can be used. At an error the break gives a backtrace on which informations from the evaluation so far are printed. We can find the forms currently being evaluated, the functions which have been called, and the variables and its values to these functions.

The editor can be used in exactly the same way as interactively. We have of course no way of undo-ing changes. A good strategy could be to use P for seeing on what structures we are working and to pretty-print the edited expression afterwards.

The editor commands can be put as arguments in editf, editp or editv, which is recommended in batch use.

```
eg  (EDITF REV 3 P 3 2 P 3 2 (1 CAR) OK)
```

10.6 File handling. When a number of functions are defined and tested there is a simple way to save their definitions on a symbolic file, which later can be loaded. For this a partitioned dataset is needed and is described in the appendix. A file has a name containing 1 to 5 characters. If we want to create the file SORT consisting of the functions quicksort, order, compare and merger, we give the global variable SORTFNS as value a list of these functions. The variable is a concatenation of the file name and FNS. The file is created by

```
(SETQ SORTFNS '(QUICKSORT ORDER COMPARE MERGER))
(MAKEFILE 'SORT 'FAST)
```

Makefile finds SORTFNS and creates the file. The file can later be loaded and the functions will then get defined by executing

```
(LOAD 'SORT)
```

In section 20 file handling is described more in detail. There are also ways to specify that we want to save global values, property-list information, S-expressions etc.

11. Some more functions

11.1 In the previous sections we have introduced a small collection of functions necessary to understand and practice LISP. There are however, more rather elementary functions which we will introduce here.

11.2 Selectq. This function is used when we depend on a value to execute different pieces of LISP code. It corresponds to the case statement in Algol 68 for example.

```
(SELECTQ p
    (p1 e1)
    ((p21 p22 p23) e2)
    (p3 e31 e32 e33)
    en)
```

p, a form, will be evaluated and its value is compared with p1, p21, p22 ... , which should be atoms and which are not evaluated. For the first p_i equal to p, corresponding forms e_i are evaluated. If no p_i is equal to p the form en is evaluated.

The first element in a sublist can either be an atom or a list of atoms. If it is a list we can have several values which execute the same piece of code. The remaining elements of the sublist are forms.

Example. Execution of different functions is dependent on the length of a list.

```
(SELECTQ (LENGTH L)
    (0 (EMPTYLIST))
    (1 (ONELIST L))
    ((2 3 4) (TWO-TO-FOURLIST L))
    (5 (FIVELIST L))
    (MORE-THAN-FIVELIST L))
```

Length is defined in Section 11.6

11.3 Progl and progn. In some situations we are only permitted to have one form, as eg, the predicat in a branch in a cond-expression and the last expression in selectq. These two functions take an arbitrary number of forms as arguments, evaluates them all and returns the following values

progl - the value of the first form

progn - the value of the last form

This corresponds to Algol's begin ... end parenthesis.

```
eg  (SELECTQ CASE
         (C1 (FOO L))
         (C2 (FIE L))
         (PROGN (FUM1 L) (FUM2 L) (FUM3 L)))
```

In the case, when the value of case is not equal to C1 or C2 the progn-expression will evaluate all three forms and return as value the value of fum3[l].

eg progl is useful when we have for example, two forms, which must be evaluated in a specific order and that we require the value from the first form. One example of this is when we want a value from a property list and the remove of that value.

```
progl[getp[x,y], remprop[x,y]]
```

In some LISP systems we are only allowed to have one form after then in a cond-expression. If we want more forms we must use progn. In a system where we have not these restrictions, we say that we have implicit progn.

11.4 Setqq. In the family of functions for assignment there is also setqq. It assumes that its two arguments shall be treated as quoted arguments. These are equivalent expressions

```
(SET 'A '(A B C))
(SETQ A '(A B C))
(SETQQ A (A B C)
```

11.5 More predicates

litatom[a]	Returns T if a is a literal atom, otherwise NIL.

```
eg  litatom[A] = T
    litatom[NIL] = T
    litatom[3] = NIL
```

listp[l] Returns *l* (=true) if *l* is a list other than NIL, otherwise it returns NIL.

```
eg  listp[(A)] = (A)
    listp[A] = NIL
    listp[()] = NIL
```

nlistp[l] Returns the opposite value as *listp*.

```
eg  nlistp[(A)] = NIL
    nlistp[(A)] = T
    nlistp[()] = T
```

neq[x,y] Returns the opposite value as *eq*.

```
eg  neq[NIL,()] = NIL
    neq[(A),A] = T
    neq["A",A] = T
```

member[x,l] Identical to *memb* but it uses *equal* instead of *eq* to check if *x* is a member in *l*

```
eg  member[A, (X A H)] = (A H)
    member[(A B),(A B (A B) C)] = ((A B) C)
    member[123.4,(4.321 123.4)] = (123.4)
```

Remember that atoms and small integers are *eq*, and that big integers, floating-point numbers, strings and lists are normally not *eq*, they are instead *equal*.

11.6 Functions for list manipulation

list[x_1,x_2, ... ,x_n] x_i is an arbitrary LISP-expression. *List* makes a list with x_i as elements. The value of *list* is the created list.

```
eg  list[A,NIL,3,(X Y)] = (A NIL 3 (X Y))
```

append[x_1,x_2, ...,x_n] x_i is a list. *Append* creats a new list of the list element of x_i. The value is the new list.

```
eg append[(A B),(Q),(X (Y))] = (A B Q X (Y))
```

In actual fact *append* makes copies on top level of x_1 to x_{n-1} and concatenates them to x_n.

A special case is when *append* gets one argument x_1. It is then copied on top level. To copy a list-structure on all levels use *copy*.

A function *nconc* (21.4) works as *append*, but does not copy any structures.

Compare and really try to understand the differences between *cons*, *list* and *append*.

```
cons[(A B),(C D)] = ((A B) C D)
list[(A B),(C D)] = ((A B) (C D))
append[(A B),(C D)] = (A B C D)
```

copy[x]	*x* is a list. *Copy* makes a copy of *x*, which is the value. All levels of *x* are copied; *x* and its copy are *equal* to each other.
remove[x,l]	*x* an arbitrary expression, *l* a list. *Remove* creates a new list, where all elements *equal* to *x* are removed. The value is the new list. eg remove[(X Y), (X Y (X Y) (X (X Y))] = (X Y (X (X Y)))
reverse[l]	*l* is a list. *Reverse* reverses the elements on top level of a list. eg reverse[(A B (C D))] = ((C D) B A)
subst[new,old,l]	*new* and *old* are arbitrary expressions and *l* a list. *Subst* creates a new list, where all expressions *old* are changed to *new* on the list *l*. The value is the new list. eg subst[A,B,(B A (B X (B X))] = (A A (A X)) subst[A, (B C),((B C) B C)] = (A B C) If the list ends with a dotted-pair, *cdr* of that pair is also changed if *cdr* of the pair is atomic. eg subst[A,B,(B A . B)] = (A A . A)
intersection[x,y]	*x* and *y* are lists. Returns a list of those elements which are members of both *x* and *y* eg intersection[(A B (C) D),((C) (A) B)] = (B (C))
union[x,y]	*x* and *y* are lists. Returns a list of those elements which are members on either *x* or *y*. eg union[(A (B C) D) ,(B C D)] = (A (B C) D B C)

last[l]	l is a list. Gives as value a pointer to the last list cell on l. eg last[(A B C D)] = (D)
length[x]	x a list or atom. Returns the number of elements on a list, or 0 if x is an atom. eg length[(A (B C)] = 3 length[ATOM] = 0

11.7 More property-list functions

deflist[atm-val,prop] atm-val is a list of two-element sublists. The first element of each sublist is the atom, on which property list under the property prop the second element is stored as value. The value from deflist is the list of atoms.

```
eg  deflist[((ONE ETT) (TWO TVA) (THREE TRE)),
                                        SWEDISH]
```

will work as

```
put[ONE, SWEDISH, ETT]
put[TWO, SWEDISH, TVA]
put[THREE, SWEDISH, TRE]
```

Extension to addprop.

addprop[atm,prop,new,flg]   See Section 5.2. If flg is T new is added first to the list, otherwise it is added at the end.

```
eg  addprop[A,B,C]         the value is (C)
    addprop[A,B,D,T]       the value is (D C)
    addprop[A,B,E,NIL]     the value is (D C E)
    addprop[A,B,F]         the value is (D C E F)
```

11.8 Assoc and sassoc.

An association list (abbreviated a-list) is a list of pairs. It can be used to hold, for example, variables and its values.

eg An association list can look like

((ETT . 1) (TVA . 2) (TRE . 3) (FYRA . 4) (FEM . 5))

assoc[x,al]	x is an atom and al an association list. The value returned is the first pair, which first element is eq to x. If no pair is found NIL is returned. eg Suppose swednumber has the above a-list as value. assoc[TRE,swednumber]= (TRE . 3) assoc[SEX,swednumber]= NIL
sassoc[x,a]	as assoc, but compares with equal.

Exercises

1. Assume we have represented a number of cards as an association list. In every pair, car is one of SPADE, HEART, DIAMOND or CLUB and cdr one of 2,3,4, ... , 10, JACK, QUEEN, KING and ACE.

   eg ((SPADE . 5) (HEART . QUEEN) (CLUB . ACE) (HEART . 5))

   In a special game we get the following points

ACE of SPADES	10	QUEEN	3
ACE of HEARTS	9	JACK	3
ACE of DIAMONDS	8	7	1
ACE of CLUBS	7	3	1
KING	5	remainder	0

   Write a function pointlist [hand], where hand is a list defined as above, which gives as value a list of the points (>0) the hand contains,

   eg pointlist[((SPADE . 5) (HEART . QUEEN) (CLUB . ACE)
   (HEART . 5))] = (3 7)

2. Define prog2[x,y], which returns y as value.

3. Define *member and *last.

4. Define *addprop, by using put and getp.

5. Define *deflist, by using put.

6. There is a system function sqcdr[l], which returns as value the first element on l and sets l to cdr[l]

   Suppose l has value (A B C)

   sqcdr[l] returns value A

   l has now value (B C)

   Define ::sqcdr.

7. For working with association lists it can be necessary to define some more auxiliary functions for manipulating them.

   a. Define the system function ::assoc.

   b. Define a function chassoc[al,a,new] which searches the first pair (a_i . v_i), such that a = a_i and changes that pair in such a way that v_i will be replaced by new.

      eg chassoc[((A . 1) (B . 2) (A . 3)), A, -1] =
      ((A . -1) (B . 2) (A . 3))

   c. Define a function repassoc[al,a], which removes all pairs (a_i . v_i), if a = a_i.

      eg repassoc[((A . 1) (B . 2) (A . 3)), A] = ((B . 2))

8. A concept very similar to association list is the free property-list. By this we mean a property-list not connected to an atom, but only as a free list structure, but containing property/value pairs.

   There is a system function get, for retrieving a value on a free property-list.

   get[freeprop, prop] gives as value the element after prop, if prop is not found, NIL is returned.

   eg get[(MOTHER ANNE FATHER JOHN),FATHER] = JOHN

   Two warnings associated with get!

   - If a value happens to be the same as prop and comes before prop on the free property-list, the correct value is not returned.

     eg get[(RELATION MOTHER FATHER JOHN MOTHER ANNE),
     MOTHER)] = FATHER

   - In some LISP systems get is used instead of getp and works on ordinary property-lists. Be careful about this!

   a. Define the function ::get.

b. Define a function putf[f,p,v], which works as put.

```
eg  putf[(MOTHER ANNE FATHER JOHN), FATHER, JIM] =
                                          (MOTHER ANNE FATHER JIM)

    putf[(MOTHER ANNE), FATHER, JIM] = (MOTHER ANNE FATHER JIM)
```

12. Arithmetic functions

12.1 In the first section we defined integers and floating-point numbers; in this section we introduce arithmetic functions. LISP is not intended to be a language for making advanced arithmetic calculations. In the functions we are allowed to use both integers and floating-point numbers. The system will make the necessary conversions.

12.2 The integers can be of two types. A small integer x, has its range

$$-2^{24} \leqslant X \leqslant 2^{24}-1$$

and a big integer x its range

$$-2^{32} \leqslant X \leqslant -2^{24}-1 \qquad \text{and} \qquad 2^{24} \leqslant X \leqslant 2^{32}-1$$

They are represented in different ways but for the user there is no real difference. The only apparent difference is that small integers have a unique representation and can be tested by eq.

The print-routine prints a floating-point number x only with a decimal point if it is in the range of

$$|x| < 10^7,$$

otherwise it is printed with an exponent.

```
eg  123.45
    1.23456E8
```

12.3 The following predicates can have arbitrary arguments.

numberp[x]	if x is a numeric atom then T else NIL
eqp[x,y]	if x is eq to y or if x and y are the same numerical atoms then T else NIL

```
eg  eqp[ATM,ATM] = T
    eqp[3.45,0.345E1] = T
    eqp[1234567,1234567] = T
```

minusp[x]	if x is negative then T else NIL
smallp[x]	if x is a small integer then T else NIL

12.4 Integer arithmetic

The following functions will give an integer as value:

iplus[x_1,x_2, ... ,x_n]	$x_1 + x_2 + x_3 \dots + x_n$. If x_i is a floating-point number it is converted to an integer. This holds for all functions below.
iminus[x]	-x
idifference[x,y]	x - y
add1[x]	x + 1
sub1[x]	x - 1
itimes[x_1,x_2, ... ,x_n]	x_1 :: x_2 :: ... :: x_n
iquotient[x,t]	x / y
iremainder[x,y]	the remainder from x/y. eg 5/2 = 1
igreaterp[x,y]	if x>y then T else NIL
ilessp[x,y]	if x<y then T else NIL
zerop[x]	if x is zero then T else NIL.

12.5 Floating-point arithmetic

The same functions as with integers are also available for floating-point numbers. They are usually spelled with an f instead of an i, fplus instead of iplus.

The arguments which are not floating-point numbers will be converted and the value returned is always a floating-point numbers. The following functions exist.

fplus, fminus, ftimes, fquotient, and fgreaterp

12.6 Conversion from integer to floating-point numbers

fix[x]	_x_ arbitrary numeric atom. Gives the integer part.
float[x]	_x_ arbitrary numeric atom. Gives corresponding floating-point number.
fixp[x]	if _x_ is an integer then T else NIL.
floatp[x]	if _x_ is a floating-point number then T else NIL.

12.7 General arithmetic

There is also a collection of functions which return an integer value if all of its arguments are integers, otherwise they convert all integer arguments to floating-point numbers and return a floating-point number. They are spelled without an _f_ or an _i_. The following functions exist.

plus, _minus_, _difference_, _times_, _quotient_, _remainder_, _greaterp_ and _lessp_.

12.8 Examples

```
plus[1, 2, 3] = 6
fplus[1, 2.5, 4] = 7.5
iplus[1, 2.5, 4, 0.8] = 7
plus[1, 2, 3.5] = 6.5
iquotient[43.2, 5] = 8
quotient[43.2, 5] = 8.640000
add1[3] = 4
add1[7.8] = 8
fix[3] = 3
float[3] = 3.0
```

12.9 There are also other arithmetic functions used for logical and shifting operations on numbers.

12.10 Hexadecimal representation

A number in hexadecimal notation is written as a @ and a number. The number consists of 0,1,2, ... , 8,9,A,B,C,D,E and F.

eg @3	3
@10	16
@12	18
@B	11
@1CE	462

12.11 We can change the base used by the system for writing numbers by using the function radix.

Input	Output
10	10
(RADIX 5)	20, the old base in the new base
10	20
4	4
5	10
(RADIX 8)	5
10	12
(SUB1 (RADIX))	7, gives the current base -1

Exercises

1. Define the function *length, see Section 10.6.

2. Define the function fak[n] which computes the facorial, n! .

3. Define the function points, which takes as argument a list of numbers, such as the list produced by pointlist in Section 10, Exercise 1, and add the numbers together.

4. Write a function diff[expr,x], which makes symbolic differentiation. This is easily solved if the expressions are represented as

$$\frac{E * F - (A + 3)}{E^2}$$

```
(QUOTIENT (DIFFERENCE (TIMES E F)
                      (PLUS A 3)
          (EXPT E 2))
```

Differentiation rules:

$$\frac{du}{dx} = 1, \text{ if } u = x$$

$$\frac{du}{dx} = 0, \text{ if } u \text{ is not a function of } x$$

$$\frac{d}{dx}(u + y) = \frac{du}{dx} + \frac{dv}{dx}$$

$$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = v\frac{du}{dx} + u\frac{dv}{dx}$$

$$\frac{d}{dx}(u/v) = (v\frac{du}{dx} - u\frac{dv}{dx}) / v^2$$

$$\frac{d}{dx}(u^n) = n \cdot u^{n-1} \frac{du}{dx} \quad \text{if } n \text{ is a constant}$$

$$\frac{d}{dx}(\sin u) = \frac{du}{dx} \cos u$$

$$\frac{d}{dx}(\cos u) = -\frac{du}{dx} \sin u$$

This process of differentiating is a recursive process. Every expression $\frac{du}{dx}$ will cause a call to diff.

Hints - Let every differentiation rule be a function, such as derplus, dersin etc. The function diff can then be a big selectq-expression which selects the right rule depending on the leading function in the expression.

5. The expressions we get as value from diff in the previous exercise must be simplified if they are to be readable. Define a function simplify, which does this. The function must take care of cases such as

   (PLUS X 0), which simplifies to X

   (TIMES X 0), which simplifies to 0

   (PLUS 1 (TIMES X 1) 3), which simplifies to (PLUS 4 X)

   Try to evaluate as much as possible.

Differentiation of

```
    (PLUS (EXPT (TIMES 3 X) 2) (TIMES 2 X)
```

can give the value

```
    (PLUS (PLUS (TIMES (EXPT X 2) 0)
                (TIMES 3 (TIMES (TIMES 2 X) 1)))
          (PLUS (TIMES X 0) (TIMES 2 1)))
```

which can be simplified to

```
          (PLUS (TIMES 6 X) 2)
```

13. Logical functions

13.1 We have earlier introduced the Boolean values true and false. In LISP they are represented as ≠ NIL and NIL respectively. There are functions for the logical connectives and, or and not.

13.2 The arguments x_i can be arbitrary LISP forms.

and[x_1,x_2, ... ,x_n] The value is NIL, if some x_i have been evaluated to NIL, otherwise it returns x_n, as a true value. Arguments past the first argument equal to NIL are not evaluated.

```
eg  and[A, 12, cons[X,NIL]] = (X)
    and[A, cdr[(X)], put[A,B,C]] = NIL
          Note that the last put is not done!
    and[] = T
```

or[x_1,x_2, ... ,x_n] The value is the first x_i, which have evaluated to a non-NIL value, otherwise NIL. Arguments past the first non-NIL value are not evaluated.

```
eg  or[NIL, cdr[(X)], car[(X)], put[A,B,C]] = X
          Note that the last put is not done!
    or[cdr1[(X)], cddr[(X Y)]] = NIL
    or[] = NIL
```

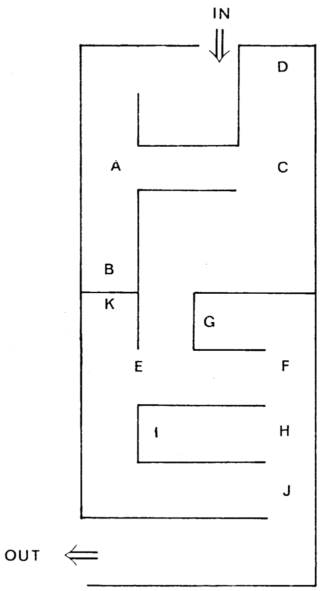
not[x] Identical to null.

```
eg  not[(A)] = NIL
    not[()] = T
```

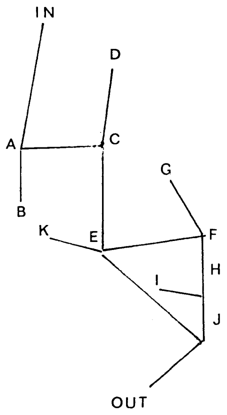
Exercises

1. Define even (see Section 9, Exercise 1) without using cond.

2. Suppose we want to solve a maze problem and find a suitable way through the maze.



This maze may be represented as a graph.



and this further represented by the following list structure

```
     ((IN A) (A IN B C) (B A) (C A D E) (D C)
      (E F K J) (F G E H) (H F I J) (G F) (I H) (J E H OUT)
                                                   (OUT J)))
```

Every sublist has as first element a node, n, and the rest of the elements are the nodes connected to n. This representation is actually an association list. If we assign the list above to MAZESTRUC

```
     cdr[assoc[B, mazestruc]] = (A)
```

will give a list of nodes, which are connected to B.

Write a function maze[mazestruc, in, out], which gives as value a list, which contains the nodes as to the way we must go from the start IN to the exit OUT

```
     eg  maze[mazestruc,IN,OUT] = (A C E F H J OUT)
```

The function need only be to find one path - not necessarily the shortest. But the way is not allowed to contain a node more than once. The following answer is invalid

```
     (IN A B A C E F H J E F H J OUT)
```

14. Function types

14.1 Among the previously defined system functions, there are some which do not follow the rules that they must have a fixed number of arguments and that they get their arbuments evaluated before entering the function. Notably the functions quote, de, and, cond, plus etc. For handling these there are different kinds of function types.

First we can separate the functions into two groups: those which have a fixed number of arguments and those which have an arbitrary number of arguments. On the other hand the functions can also be separated into those with arguments evaluated before entering the function and those which receive their arguments unevaluated.

	Gets its arguments evaluated	Gets its arguments unevaluated
Fixed number of arguments	eval-spread	noeval-spread
Arbitrary number of arguments	eval-nospread	noeval-nospread

Examples:

eval-spread	car, cons, put, union, difference
noeval-spread	setqq
eval-nospread	list, append, plus
noeval-nospread	quote, and, cond, selectq

A function like setq, where only the second argument will be evaluated, has its argument unevaluated but self-evaluates its second argument by making a call to the evaluator eval. The function-type of setq is therefore noeval-spread.

14.2 The user can introduce these different types by doing

(DE FOO (X Y Z) ....)	for an eval-spread function
(DE FOO L ....)	for an eval-nospread function
(DF FOO (X Y Z) ....)	for a noeval-spread function
(DF FOO L ....)	for a noeval-nospread function

The function de is used to define an eval function and df is used for a noeval function. If the argument list is an atom the function is a nospread and if it is a list of atoms the function is a spread.

14.3 Examples

a. (DE FOO L (LIST (CADR L) (CAR L)))

If we call

```
(FOO 'A 'B 'C 'D)
```

the arguments will all be evaluated and the list of arguments, (A B C D), will be bound to L and the function body will be evaluated and return (B A) as value.

b. (DF FIE (X Y Z) (LIST Y Z X))

If we call

```
(FIE A B C)
```

the arguments will be bound directly without any evaluation to the variables in the argument list.

```
X  gets value  A
Y  gets value  B
Z  gets value  C
```

The evaluation of the function body will return (B C A) as value. Even if we call

```
(FIE 'A 'B (CAR '(A B)))
```

no evaluation will occur. The value in this case is

```
((QUOTE B) (CAR (QUOTE (A B))) (QUOTE A))
```

c. (DF FUM L (LIST (CADR L) (CAR L)))

If we call

```
(FUM A B C D E F G)
```

the list of arguments, unevaluated, will be bound to L and the value returned is (B A).

14.4 When we have a spread function we can call that function with an arbitrary number of arguments. If they are too few the remaining variables in the argument list are bound to NIL and if they are too many they will in the eval case be evaluated and then dropped. In the noeval case they are only dropped.

```
eg  cons[A] = (A)
```

14.5 There is also a type called half-spread. The argument list ends then with a dotted-pair.

```
eg  (DE FOO (X Y . Z) (LIST Y Z X))
```

If we call

```
(FOO 'A 'B 'C 'D 'E)
```

the arguments will be evaluated and the following bindings will appear

```
X  gets the value  A
Y  gets the value  B
Z  gets the value  (C D E),
```

The value returned is then (B (C D E) A)

We can of course also define a noeval function as a half-spread.

14.6 When de or df defines a function the definition is put in the function cell of the atom, corresponding to the function name.

de and df puts a lambda and nlambda respectively, in the definition, marking if the definition is of eval or noeval type.

```
(DE FOO (X Y) (LIST Y X))
```

will put the expression

```
(LAMBDA (X Y) (LIST Y X))
```

in FOO's function cell.

```
(DF FIE L (LIST (CAR L) (CADDR L)))
```

will put

```
(NLAMBDA L (LIST (CAR L) (CADDR L)))
```

in FIE's function cell.

We call then a function definition like this a lambda or nlambda-expression.

14.7 There are more functions which can be used on function definitions.

putd[fn,expr]	Places expr in the function cell of fn. This is the most primitive way of defining a function. The value is fn. eg putd[FOO,(LAMBDA (X) (FIE X))]
getd[fn]	Gets the function definition for fn. eg getd[FOO] = (LAMBDA (X) (FIE X))
movd[from,to,copyflg]	Moves the definition of from to to. If copyflg = T a copy of the definition in from is used. eg movd[FOO,FIE,T] A copy of getd[FOO] is placed in FIE's function cell. movd[CAR,KAR] A synonym to car is defined.
movdqq[from,to,copyflg]	A noeval type of movd.
define[x]	Define is the normal function in some LISP systems to define functions. In INTERLISP/360-370 it is more convenient to use de and df. Define has a rather complicated definition, the interested reader can check further with the LISP manual.

```
eg  (DEFINE '((FOO (LAMBDA (X Y) (LIST X Y)))
              (FIE (LAMBDA L (CAR L)))
              (FUM (X Y) (CONS Y X))))
```

In the last definition define inserts a lambda.

savedef[fn]	Saves the definition of fn under the property EXPR,CODE or SUBR depending on whether the function is defined as a list expression, compiled or hand-coded in machine code. If fn is a list every definition is saved.
unsavedef[fn,prop]	Restores the definition of fn from prop. If prop is not given it searches for EXPR, CODE or SUBR, in that order.

fntyp[fn]	Fn is a function name or a function definition. It returns the function type, described by the following

list	compiled	machine coded
EXPR	CEXPR	SUBR
FEXPR	CFEXPR	FSUBR
EXPR*	CEXPR*	SUBR*
FEXPR*	CFEXPR*	FSUBR*

The prefix F indicates noeval and the suffix * indicates nospread.

eg car is a machine-coded eval-spread function

fntyp[CAR] = SUBR

append is a compiled eval-nospread function

fntyp[APPEND] = CEXPR*

(DF FOO L ...) will define foo as a list-structured noeval-nospread function

fntyp[FOO] = FEXPR*

There are some more functions working on function definitions. They are described in detail in the manual and we will only give a short description here.

- suberp, ccodedp and exprp, predicates for testing function type as described by fntyp.
- argtypes, gives the argument-types of a function.
- nargs, gives the number of arguments to a function.
- arglist, gives the argument-list of a function.

14.8 When we are redefining a function a message is written

(FOO REDEFINED)

The old definition is then saved by savedef. We can restore the old version by

(UNSAVEDEF 'FOO)

If we redefine an already redefined function the oldest version will remain on the property-list.

14.9 If we define a function as noeval, and we want to evaluate the arguments on our own, we can use the function eval. This function is LISP's evaluator which takes as input a form, evaluates the form and returns a value. Eval is an eval-spread function.

```
eg  (EVAL '(CAR '(A B C))) = A
    (SETQQ FORM (CONS NIL NIL))
    (EVAL FORM) = (NIL)
```

The use of eval is shown in the examples in Section 14.10, following.

14.10 The following examples illustrate the use of the different function types.

a. Define ::setq and ::setqq from set

```
(DF ::SETQ (VAR VAL) (SET VAR (EVAL VAL)))
(DF ::SETQQ (VAR VAL) (SET VAR VAL))
```

b. Define ::list

```
(DE ::LIST L L)
```

c. Define ::de. We assume de does not exist and must therefore use putd.

```
(PUTD ::DE '(NLAMBDA (FN . L)
            (PUTD FN (CONS 'LAMBDA L))))
```

d. Define ::or.

```
(DF ::OR L (OR1 L))
(DE OR1 (L) (COND ((NULL L) NIL)
                  ((EVAL (CAR L)))
                  (T (OR1 (CDR L)))))
```

e. Define ::quote.

```
(DF ::QUOTE (L) L)
(DF ::QUOTE L (CAR L))
```

Exercises

1. Define the function ::df.

2. Define the function ::append, as described in Section 11.6.

3. Define the function /, which is used to quote a list, so that we can write (/ A B C D) which means '(A B C D).

4. Define the function *and*.

5. Define the function *selectq*, as described in Section 11.2.

6. You could quite easily introduce other control statements from other programming languages to LISP such as if-then-else and do-until. Define a function if, by which you can write statements like

```
(IF (LESSP N 10) THEN (SETQ N (SUB1 N)) (FOO N) ELSE (FIE N))
```

   Define also a function do, by which you can write

```
(DO (SETQ N (SUB1 N)) (FOO N) UNTIL (ZEROP N))
```

   If N is 2, foo will be called with arguments 2, 1 and 0.

15. Variable bindings

15.1 A variable in LISP can have two kinds of values.

- A global value. It is stored in the atom's value cell. The value cell is initialized to NOBIND. (See Section 2.3). The value can be set by the function set, (setq, setqq), and retrieved by giving its name.

  ```
  (SETQ PI 3.14)
  PI              will give the value 3.14
  ```

  The implementation is done in such a way that car of an atom is this value cell.

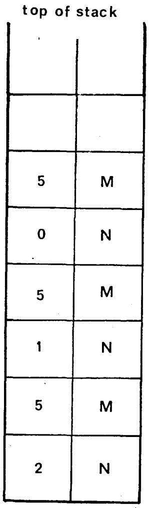
  ```
  (CAR 'PI)       will also give the value 3.14
  (CAR 'FI)       will give the value NOBIND
  FI              gives an error, U.B.A (UnBound Atom).
  ```

  There is also another function rplaca (replace car) which can be used to set a global value. See later in this section.

- A binding value. When the variable occurs in an argument list (then called lambda-variable) in a function or in a prog variable list, (see Section 16), at function call, the variable and its associated value will be put on a stack. This stack is called parameter stack. This is, of course, done to enable recursive calls to be made. Fixed location for this variable is not possible. At return from a function the variable and the value are removed from the stack.

```
eg  (DE FOO (N M) (COND ((ZEROP N) 1)
                        (T (TIMES M (FOO (SUB1 n) M)))))
```

Suppose we evaluate foo[2,5]. At the point when zerop[n] is true the stack appear thus



A variable on the stack can obtain a new value by using set (setq, setqq)

```
eg (DE FOO (N) (COND ((MINUSP N) (SETQ N (MINUS N)))) ... )
```

15.2 What happens if a variable has both a global and binding value? When binding a value by set (setq, setqq) the system searches the stack first for the variable, and if the variable is there it will be rebound, otherwise the variable's global value is set. The same procedure occurs when the system shall retrieve a value.

To be sure to change only the global value the function rplaca can be used.

rplaca[x,y] If x is an atom, the global value of this atom is set to y. Rplaca is described in more detail in Section 21.

```
eg  (SETQ N 0)
    (SETQ M 0)
    (DE FOO (N) (COND ((ZEROP N) 'GLOBAL) (T 'BINDING)))
    (FOO 3)                will return BINDING
    (DE FIE (M) (SETQ N 10) (SETQ M 10))
    (FIE 1)
    N                      has now the value 10.
    M                      has still the value 0.
```

```
(DE FUM (N M) (RPLACA 'N 5) (RPLACA 'M -5) (PLUS N M))
(FUM 2 -2)            returns 0
N                     now has the value 5
M                     now has the value -5
```

Notice here how the function rplaca can be used for setting a global value, independent of the contents of the parameter stack.

```
(DE GUM (N M) (SETQ M (ADD1 (CAR 'M))) (PLUS M N))
(GUM 5 10)            returns the value 1. m was rebound to
                      the global value of m added by one.
M                     still has the value -5
(CAR 'N)              still has the value 5
```

Note the use here of car to retrieve a global value.

15.3 A noeval version of rplaca is rpaqq for setting a global value.

```
eg  (RPLACA 'VAR 'START)
    (RPAQQ VAR START)
```

15.4 Free variables

A variable in a function definition which is not a lambda or prog variable is called a free variable in that function.

```
eg  (DE FOO (X Y) (FIE X))
    (DE FIE (X) (FUM X Y))
```

y is free in fie.

16. Prog

16.1 Up to this point we have not been able to use LISP in the more conventional way of writing programs - as a sequence of statements and with gotos for controlling the flow between them. This is possible to do in LISP too, but we have tried to avoid this so far, thus enforcing the user to be aquainted primarily with recursion. Only too often it happens that a beginner writes LISP with the prog feature alone. The recursion is not expensive, and programs written recursively tend to be more readable than if written in the iterative way.

16.2 The notation for prog is

```
        (PROG varlist expression
                      expression

                      expression)
```

Varlist is a list of variables which will be local in this prog. The list contains either atoms or sublists, where a sublist is used to initialize a variable. As default the variable initializes to NIL. These variables will be bound on the parameter stack - see Section 15.1.

```
    eg  (PROG (A B (X 10) (Y (CAR '(A B)))) ...)

        A  is initialized to  NIL
        B  is initialized to  NIL
        X  is initialized to  10
        Y  is initialized to  A
```

The evaluation of all forms is done before the binding of the variables on the stack.

eg Suppose X has the value 10

```
        (PROG ((X 5) (Y X)) ... )
```

will initialize X to 5 and Y to 10, the prog variable X has not yet been bound on the stack.

Expression can be either an atom which is then interpreted as a label or a list and is then a form which will be evaluated.

16.3 There are two special functions for prog.

return[expr]	Makes a return from the prog and the value of expr will be the value from the prog.
go[l]	A noeval function. Will transfer the control in the prog to the label l. l must exist in the last entered prog, otherwise an error message will occur, so we are not allowed to jump out from a prog to a label in another prog.

16.4 Example

Define the function ::length.

```
(DE ::LENGTH (L)
     (PROG ((N 0))
     LOP
     (COND ((NULL L) (RETURN N)))
     (SETQ L (CDR L))
     (SETQ N (ADD1 N))
     (GO LOP)))
```

Exercises

1-6 Define the functions from exercises 1-6 in Section 9.
Interaction should be used where possible instead of recursion.

17. Evaluating functions

17.1 As seen in the previous section it is necessary to be able to call the evaluator (interpreter). This was used to evaluate arguments in a noeval function. There are two main functions, eval and apply, by which the user can call the interpreter.

17.2 eval[form] The form will be evaluated. Notice that eval itself is of eval-type, so its argument is first evaluated.

```
eg  (EVAL '(CONS T T)) = (T . T)

    (SETQQ FORM (PLUS 3 5))
    (EVAL FORM) = 8
    (EVAL 'FORM) = (PLUS 3 5)
```

e[form] Noeval-nospread version of eval.

```
    (E (CONS T T)) = (T . T)
```

evala[form,alist] Simulates a-list evaluation as in LISP 1.5.

apply[fn,args] The function fn will be applied to the arguments in args. Apply is of eval type but observe that arguments in the argument-list args are not evaluated.

```
eg  (DE FOO (X Y) (CONS Y X))
    (DF FIE (X Y) (LIST Y X))
    (FOO '(B) 'A) = (A B)
    (FIE (B) A) = (A (B))
    (APPLY 'FOO '((B) A)) = (A B)
    (APPLY 'FIE '((B) A)) = (A (B))
```

apply*[fn,arg_1, ... , arg_n] is equivalent to apply[fn,list[arg_1, ... , arg_n]]

```
eg  (APPLY* 'FOO '(B) 'A) = (A B)
```

17.3 When the form is a list, car of that list is the function which will be applied to the arguments.

car of form can be

- a function-name, ie car, foo (if foo is, or will be defined as a function)
- a lambda or nlambda expression,

```
((LAMBDA (X Y) (CONS Y X)) '(I S P) 'L) = (L I S P)
((NLAMBDA L (LIST (CADR L) (CAR L))) A B C D) = (B A)
```

- a funarg expression, described in Section 28.
- a function indicator - see the LISP manual.

In the original version of LISP 1.5 car of form could be of arbitrary form. If it was not a function name or a lambda or nlambda expression it was evaluated. This will in INTERLISP give an error, but we can use apply or apply:: instead.

eg Suppose we have done

```
(PUT 'A 'FN 'CAR)
```

The form

```
((GETP 'A 'FN) '(A B C))
```

is not permissable, but we can instead write

```
(APPLY:: (GETP 'A 'FN) '(A B C))
```

17.4 In the differentiation exercise (Section 12, Exercise 4) we proposed a solution where we had a main function, diff, controlling the calls to the various sub-functions. Each sub-function corresponded to a differentiation rule. If we now want to extend the set of rules this can be achieved by adding code in diff and translating the rule to a LISP function.

Another solution to this would be to let each rule be a lambda-expression in the same way as before, and let these expressions be stored on property-lists.

eg The rule

$$\frac{d}{dx}(u-y) = \frac{du}{dx} - \frac{dy}{dx}$$

is translated to

```
(LAMBDA (EXPR X)
    (LIST 'DIFFERENCE
          (DIFF (CADR EXPR) X)
          (DIFF (CADDR EXPR) X)))
```

and is stored by

```
    (PUT 'DIFFERENCE 'DIFFRULE '(LAMBDA (EXPR X) ... ))
```

This simplifies diff and can now be defined as

```
 (DE DIFF (EXPR X)
   (PROG (RULE)
         (RETURN (COND ((EQ EXPR X) 1)
                       ((ATOM EXPR) 0)
                       ((SETQ RULE (GETP (CAR EXPR) 'DIFFRULE))
                            (APPLY:: RULE EXPR X))
                       (T EXPR)))))
```

We can now extend the number of rules with ease, without changing any code in the existing functions.

17.5 Getdflt. Another example where it can be useful to have "functions" stored on property-lists is to handle default routines, for calculating a property value when it is not explicitly given.

Suppose we have the property height. If the height for a person is not given we want a default value (176 for boys or 166 for girls) to be stored.

A general solution to this is to have a function getdflt defined as

```
    (LAMBDA (ATM PROP) (OR (GETP ATM PROP)
                           (APPLY:: (GETP PROP 'DFLT) ATM)))
```

We also define a default routine for height

```
    (PUT 'HEIGHT 'DFLT '(LAMBDA (ATM)
                           (PUT ATM
                                'HEIGHT
                                (COND ((EQ (GETP ATM 'SEX) 'BOY) 176)
                                      (T 166>
```

If we have stored the fact that EVA is a girl

```
    (GETDFLT 'EVA 'HEIGHT)
```

will first make a getp[EVA,HEIGHT], which will give NIL as value and then apply the default routine height to EVA which will store a default-value, calculated to 166, and return it as value. The next time the same expression is evaluated the height is already stored.

For every property p we want default routines, we had to store a dflt property on p's property-list and then use getdflt instead of getp. If a dflt routine is missing NIL is returned.

(APPLY* NIL arg_1... arg_n) will always return NIL

Exercises

1. Define a function calc[op,a,b], where op is a functional argument and which specifies the arithmetic operation done by calc on the arguments a and b.

```
eg  calc[PLUS, 10, 20] = 30
    calc[(LAMBDA (X Y) (COND ((GREATERP X Y) X) (T Y)))),
                                        10, 20] = 20
```

2. Run the following examples on the computer and study the results. Can you explain them?

```
(APPLY 'SET '(A B))
(APPLY 'SETQ '(E F))
(APPLY 'SETQQ '(I J))
```

3. Define a function first[l,fn], which gives the first element x on the list l, satisfying fn[x].

```
eg  first[(A B 1 C 2 3) , NUMBERP] = 1
```

18. Map functions

18.1 The map functions are a collection of useable LISP functions which are characterised in that they work on a list of elements and that they on each element apply a function. This function is an argument of the map function. For quote-ing a functional expression there is a function function which uses the funarg feature described in the LISP manual.

18.2 function[fn,freevars]	If freevars is NIL then it is identical to quote, but it helps the compiler to show that this is a functional argument. When freevars is ≠ NIL it is a list of variables which presumably are free in fn. A funarg expression will then be created, but this is further discussed in Section 28.
mapcar[mapx,mapfn1,mapfn2]	If mapfn2 is NIL, then mapfn1 - which should be a function - is applied to every element on the list mapx, and then returns a list of those values computed.

```
(MAPCAR '(1 2 3 4) (FUNCTION ADD1)) =
                          (2 3 4 5)

(MAPCAR '(CAR APPEND MAPCAR) (FUNCTION
       FNTYP)) = (SUBR CEXPR* CEXPR)

(MAPCAR '(1 15 5 25 30 10)
        (FUNCTION (LAMBDA (X)
            (AND (GREATERP X 12)
                 (LESSP X 28))))) =
                   (NIL T NIL T NIL NIL)
```

If mapfn2 is provided, then instead of using cdr for computing the next element of mapx, mapfn2 is used.

```
eg  (MAPCAR '(1 2 3 4 5 6)
            (FUNCTION ADD1)
            (FUNCTION CDDR)) = (2 4 6)
```

mapc[mapx,mapfn1,mapfn2]	Identical to *mapcar*, but it returns the value NIL and does not build a list and use *cons*'es.
maplist[mapx,mapfn1,mapfn2]	Instead of applying *mapfn1* to an element in *mapx*, it is applied to successive tails of *mapx*. eg maplist[(A B C D E), LENGTH] = (5 4 3 2 1) length[(A B C D E)] length[(B C D E)] length[(C D E)], etc are computed.
map[mapx,mapfn1,mapfn2]	Identical to *maplist*, but it returns NIL instead.
map2car[mapx,mapy,mapfn1,mapfn2]	Identical to *mapcar*, but *mapfn1* is a function of two arguments and mapfn1[car[mapx],car[mapy]] is computed at every step. eg map2car[(A B C D E), (X B Y D E),EQ] = (NIL T NIL T)
map2c[mapx,mapy,mapfn1,mapfn2]	Corresponds to *mapc*.
every[mapx,mapfn1,mapfn2]	If the result of applying *mapfn1* to every element of *mapx* is *true*, the value T is returned. If the result is NIL, *every* immediately returns the value NIL. eg every[(A B C D), ATOM] = T every[(10 20 5 15 25), (LAMBDA (X) (GREATERP X 8))] = NIL
some[mapx,mapfn1,mapfn2]	For the first result of applying *mapfn1* to the elements of *mapx*, which is *true*, *some* will return a *true* value. If all results were *false*, *some* returns NIL. eg some[(A NIL (X Y), Z),LISTP] = ((X Y) Z) the value returned is the remainder of the list where *car* of that list is the element which gave a *true* value.

```
every[(10 20 5 15 25)
      (LAMBDA (X) (GREATERP X 8))] = NIL
```

Memb can be defined as

```
(DE *MEMB (X L)
   (SOME L (FUNCTION (LAMBDA (Y)
                 (EQ X Y)))))
```

There are more map functions in the system - if interested see the LISP manual.

Exercises

1. Define the functions *map, *mapcar, *map2c and *every.

2. Define a function square, which computes the square of each element of a list

   eg square[(1 2 3 4 5 6)] = (1 4 9 16 25 36)

3. Suppose we have stored under the property SONS a list, sons. Define a function storefather[a], where a is an atom with the above property. Storedef will store the value a under the property FATHER on the property-lists of every atom on the list sons.

   eg put[JOHN, SONS, (JIM TIM PIM)]

   Storefather[JOHN] will then make

   put[JIM,FATHER,JOHN]

   put[TIM,FATHER,JOHN]

   put[PIM,FATHER,JOHN]

4. Define pair[x,y], which makes an association list of x and y. We assume that x and y have the same length.

   eg pair[(A B C D), (1 2 3 4)] = ((A . 1) (B . 2) (C . 3) (D . 4))

5. Define collectpairs[al,a], where al is an association list and a an atom which returns a list with all pairs, whose car is eq to a.

   eg collectpairs[((A . 1) (B . 2) (A . 3)), A] = ((A . 1) (A . 3))

19. I/O functions

19.1 LISP has the advantage in that we can start using it - writing complex programs - without knowing anything about I/O. The system's read-eval-print loop takes care of this. There are of course, functions that enable the user to specify his own I/O if he wishes to make a formated output - ie, a table - or if he wants to I/O another media other than the terminal - ie, a file on a disc store. Yet another very important feature of LISP is that I/O is defined for list structures. In languages like PL/L, Simula etc, we can not simply print a list structure.

19.2 I/O in LISP is made from, or to, a file. Normally it is the terminal when using it interactively, or the card reader (remote job terminal) and printer, when it is used in batch. The system knows when it is used interactively or in batch, and these files are the initial primary files. T is used to indicate these files. All I/O functions have as optional argument, the file. If it is omitted or NIL the primary file is taken. All files must be opened before they can be used, except T which is always open.

19.3	input[file]	File is the new primary input file. Its value is the old file. If file is NIL the current primary file is returned.
	output[file]	Same as input, but works on output primary file.
	infile[file]	File is opened for input, and the input primary file sets to file. The old input primary file is returned as value.
	outfile[file]	Same as infile, but opens an output file.
	closef[file]	Closes file.
	closeall[]	Closes all opened files.

openp[file,type] If type = NIL tests if file is opened
INPUT tests if file is opened for input
OUTPUT tests if file is opened for output.

Returns file if the test succeeds, and NIL otherwise. File = NIL will return a list of all open files.

eg To open a file only, without setting the primary files

input[infile[file]].

19.4 When the file is on disc store it must follow some rules. The file name must be 1 to 5 characters long, and an optional generation number can be used. We will deal in more depth with file handling in the next section.

eg Valid file name FOO
FIE#03

19.5 Input functions

read[file] Reads one S-expression (actually atom, string or list) from file, which is returned as value. The same function is used by the LISP read-routine, so the rules concerning delimiters given in Section 1 are also valid here.

ratom[file] Reads in the next atom from file. Break characters as ( ) < > " and ' will also be interpreted as atoms. See further 19.7.

readc[file] Reads next character from file.

eg Suppose the following character string is in the input buffer

ABC "C" (X)'A%<

successive read reads

ABC, "C", (X), 'A, and < 'A will be translated to (QUOTE A)

successive ratom reads

ABC, ", C, ", (, X,), ', A and <

successive readc reads

A,B,C, ,",C,", ,(,X,), ' ,A, ,% and <

There are some more input functions in the system, described in detail in the LISP manual, we give a very brief description here.

- rstring reads a string
- ratoms reads a sequence of atoms by ratom
- readp looks in the input buffer to see if there is anything there
- readline reads a line from terminal
- peekc looks at next character in input buffer, but does not read it.

19.6 Output functions

print[x,file]	Prints x on the file file followed by a new line. Value returned is x. The expression printed contains the escape character (%) and the string separator ("). An expression printed by print, can later be read properly by read.
prin1[x,file]	Prints x on the file file, without % and ".
prin2[x,file]	Prints x on the file file, as print but without making a new line.
terpri[file]	Makes a new line.
spaces[n,file]	Prints n blanks on file.

```
eg  (PROGN (PRINT 'ADAM) (PRINT ' "JOHN"))
    outputs
    ADAM
    "JOHN"

    (PROGN (PRIN1 'ADAM) (SPACES 1)
                (PRIN1 ' "JOHN") (TERPRI))
    outputs
    ADAM JOHN

    (PROGN (PRIN2 'ADAM) (SPACES 1)
                (PRIN2 ' "JOHN") (TERPRI))
    outputs
    ADAM "JOHN"
```

eject[]	new page on line printer
printlevel[n][1]	On file T (terminal and printer) the printing can be controlled by the depth in the structures. n states that n single left parenthesis will be printed, below that all lists will be printed as &. The value is the old setting. n=NIL gives the current setting. eg printlevel[2] Suppose l is (A (B C (D (E) F) G (H)) K) print[l] would print (A (B C & G &) K)
linelength[n,file]	Sets the length of the line on file Can be used both for input and output files. See further the LISP manual.
prettyprint[x]	x is a list of functions or a variable whose value is a list of functions. The definitions of the functions will be printed in pretty format on the primary output file.
pp[x]	Noeval-nospread function. Prints the list x of functions with prettyprint on the file T. Comments are printed as ::COMMENT::.
pp::[x]	As pp, but comments are printed. Short comments are printed to the left and long comments are printed within the function, but separated from the code by blank lines. eg (PRETTYPRINT '(FOO FIE FUM)) or (SETQQ PRFNS (FOO FIE FUM)) (PRETTYPRINT 'PRFNS) or (PP:: FOO FIE FUM)
printdef[l]	Prettyprints an arbitrary list structure. eg (PRINTDEF (GETP 'A 'X))

19.7 There is a standard set of break characters and separators in the system.

A break character will delimit atoms and the character itself is also interpreted by ratom, as an atom. They are ( ) < > and ",

[1] The print-level can also be set by attention-P, see further 30.5.

A separator will also delimit atoms, but they are not interpreted by ratom, as an atom. They are   (blank) and ↲ (line feed).

There are functions by which we can set and use own delimiting characters

setsepr[lst,flg] — The lst, a list of character codes (EBCDIC codes), sets new separator characters. When flg is NIL lst replaces the old set of separator characters. For other actions of this function see further the LISP manual

setbrk[lst,flg] — Same as setsepr, but sets break characters.

eg In an algorithmic language there can be formulas like

```
A=B+C/D;  A=BETA+C/DELTA;
```

If we are to read this expression and immediately get the parts of the statement we can make

```
setbrk[(126 78 96 97 92 94)]
```

and then have a loop making ratom until ; is read in. The numbers are the internal character codes for

= + - / :: and ;

getsepr[] — gives a list of current separators.

getbrk[] — gives a list of current break characters.

character[code] — gives the character with EBCDIC code code.

eg character[78] = +

character[94] = ;

chcon[chrs] — Returns a list of the EBCDIC-codes for the characters in chrs.

eg chcon[+;] = (78 94)

Only the functions ratom, uread and prin3 are affected by the user's set of break and separator characters. Ratom is described in Section 19.5.

uread[file] — user read. Same as read but uses the user's set of break and separator characters. If ( ) < and > are included in the break characters it will read lists as read but spliting atoms containing break or separator characters.

eg Suppose blank and comma are separator characters and ( ) + ∷ are break characters. They are set by

```
setsepr[(64 107)]

setbrk[(77 93 78 92)]
```

If the input stream is

```
(FOO X+Y,Z,(FIE Y∷Z))
```

uread gives the list

```
(FOO X + Y Z (FIE Y ∷ Z))
```

prin3[l,file] Prints l, on file, so it can be read again by uread. It uses the user's break and separator characters to determine when to insert %-s.

19.8 Example

INTERLISP/360-370 works normally in eval mode, which means that we write expressions which the top-loop gives directly to eval. There are other top-loops in other LISP systems and the most common is the apply mode.[1] This mode reads two expressions, the first a function, and the second an argument list, and gives this to apply.

```
eg  CAR ((A B C D)) = A

    CONS (X (A B)) = (X A B)

    DE (FOO (X) (CAR X)) = FOO
```

This can be done by defining a function applyloop.

```
(DE APPLYLOOP NIL
    (PROG NIL
          LOP
          (PRINT (APPLY (READ T) (READ T)) T)
          (GO LOP)))
```

The above function can be used interactively, but in batch we can use

```
(DE APPLYLOOP NIL
    (PROG NIL
          (PROG (CLA CLB)
          LOP
          (PRIN1 ' "---------------------------" T)
          (TERPRI T)
          (SETQ CLA (CLOCK 2))
          (PRINT (APPLY (PRINT (READ T) T) (PRINT (READ T) T)) T)
          (SETQ CLB (CLOCK 2))
          (SPACES 23 T)
          (PRIN1 (IDIFFERENCE CLB CLA) T)
          (SPACES 1 T)
          (PRINT 'MS T)
          (GO LOP)))
```

[1] In some LISP 1.5 systems the top-loop works in an evalquote mode, which nearly corresponds to the apply mode.

This will output

```
---------------------------

CAR

((A B C))

                         25 MS

---------------------------

CONS
```

The call clock[2] gives a time in ms. The difference between two such calls gives the computing time in ms between these two calls.

There are two weaknesses in the above loop. What happens if an error occurs? We then return to the top loop. In the LISP manual there is a function errorset, which prevents us in such cases. See Section 25. The other weakness is that it is no normal way to enter the top loop again. This can be done by defining that a return will be made when it reads a function equal to NIL, or something similar.

19.9 Actually the LISP top loop works so you can enter expressions given either in eval mode or in apply mode. The top loop is a function lispx[1] and the function itself decides if the expression will be given to eval or apply. Simply, if it reads an atom followed by an expression it is taken as input for apply, otherwise to eval.

```
-(CONS 'A '(B C))

(A B C)

-CONS (A (X Y))

(A X Y)

-(CAR '(A B)) (SETQ VAL '(X Y))

A

(X Y)

-VAL

(X Y)
```

[1] The function lispx can be redefined, so make sure how your version behaves.

Exercises

1. Write a function pascal[n], where n is a number between 0 and 10, which constructs Pascal's triangle and writes it out as a triangle.

   pascal[6] will print

```
                  1
               1     1
            1     2     1
         1     3     3     1
      1     4     6     4     1
   1     5    10    10     5     1
1     6    15    20    15     6     1
```

   A number within the triangle is the sum of the two numbers above.

2. Define a function algolscan, which can read statements written in Algol and which converts it to some internal list form.

```
eg  (ALGOLSCAN)
    A:=B+C;
    IF X>10 THEN L:=10 ELSE BEGIN L:=5; GOTO H END;
    ENDALGOL
```

   The internal form could be a list of the different syntactic entities. The above example gives the list

```
(A := B + C ; IF X > 10 THEN L := 10 ELSE BEGIN
L := 5 ; GOTO H END ;)
```

20. File handling

20.1 The system provides a rather advanced file-handling feature. To create a file in LISP's sense is to write out the symbolic notation of LISP expressions by the system's print-routines. This means that an atom is written as its character string and a list, is written by parentheses etc. System's read-routine can then read it in again and create the internal structures.

A file has a name (1-5 characters) and a generation number. Every time a file is created with the same name the generation number is updated. A file can be retrieved either by giving only the name - which then gives the latest generated file - or by giving the name and a generation number. This makes it possible to automatically have back-up on the files we have. A file must be opened before it can be used.

```
eg  (OUTFILE 'FOO)         FOO is opened and is set to be the primary
                           output file
    (PRINT '"THIS IS WRITTEN ON FILE FOO")
    (PRINT '(THIS IS ALSO WRITTEN ON FOO))
    (CLOSEF 'FOO)          FOO is closed and  FOO≠00[1]  is created
    (INFILE 'FOO)          FOO≠00  is opened again as input file
    (READ)                 reads "THIS IS WRITTEN ON FILE FOO"
    (READ)                 reads (THIS IS ALSO WRITTEN ON FOO)
    (CLOSEF 'FOO)          FOO is closed again
    (OUTFILE 'FOO)
    (PRINT '"THIS IS WRITTEN ON A NEW FILE")
    (CLOSEF 'FOO)          FOO is closed and  FOO≠01  is created
    (INPUT (INFILE 'FOO))     FOO≠01  is opened, but T is still
                              the primary file
    (INPUT (INFILE 'FOO≠00))      Also  FOO≠00  is opened
```

[1] A file with generation number consists of file name, ≠ and a two-digit number

eg   FOO≠13

```
(READ 'FOO≠00 )      reads "THIS IS WRITTEN ON FILE FOO"
(READ 'FOO)          reads "THIS IS WRITTEN ON A NEW FILE"
(CLOSEF 'FOO)
(CLOSEF 'FOO ≠00)
```

20.2 The actual implementation with regards to the operating system is described in the LISP manual. In an OS/360 environment we need a partioned dataset. A LISP file will be stored as a number in that dataset. To delete and compress files we must use IBM's utility programs. In the appendix to the current JCL is a description for this.

20.3 Makefile and load

There is a function makefile which makes the use of files very simple. This function takes a number of variables, functions, properties etc and writes out the function definitions, variable values, property values etc. in such a way that when they are loaded in again the functions will be defined again and the variables and properties obtain its old values. An example illustrates its use

```
(SETQQ FOOFNS (FUM GUM HUM))
(SETQQ FOOVARS (FIE GIE HIE (PROP FOOFLG A B C D)))
```

These two global variables[1] describe that we want to create a file foo, which contains the functions fum, gum and hum and the variables fie, gie and hie and the values under the property fooflg for the atoms a, b, c and d

```
(MAKEFILE 'FOO 'FAST)
```

All definitions and values are written on file foo.

When we later make

```
(LOAD 'FOO)
```

the file will be read and all definitions and values re-stored again. The global variables foofns and foovars were also stored, so they can be used if we want to up-date the file.

[1] The global variables are created by the file name, without generation number, concatenated to FNS and VARS respectively.

we can then do

```
(SETQQ FOOFNS (CONS FIENEW FOOFNS))
(SETQQ FOOVARS (REMOVE 'GIE FOOVARS))
(MAKEFILE 'FOO 'FAST)
```

A new generation of foo is created.

fileFNS is a list of function names.

fileVARS is a list of commands and the most common ones are

- if atomic it defines a variable
- (PROP property $atom_1$ ... $atom_n$). Defines values on $\underline{atom}_i$ under property. If property = ALL, it defines all values on a property list, but no system properties. If property is a list it defines values for each property on that list.
- (P $sexpr_1$ ... $sexpr_n$) Defines S-expressions which will be printed on the file. This expression will be evaluated at load-time.
- (E $sexpr_1$ ... $sexpr_n$) each S-expression $\underline{sexpr}_i$ will be evaluated and the value will be printed on the file.
- (FNS fn_1 ... fn_2) Defines functions. This can be useful if we want to make some computation, eg with the P-command, before the functions will be defined when loaded.

  eg
  ```
  (RPAQQ FOOFNS (FOO1 FOO2))
  (RPAQQ FOOVARS ((P (MOVDQQ FIE FUM)) (FNS FIE)))
  (MAKEFILE 'FOO 'FAST)
  (LOAD 'FOO)   will first define foo1 and foo2, then
                make the move and at last define fie.
  ```

- (VARS var_1 ... var_n) If $\underline{var}_i$ is atomic it is printed so it will be set to the global value of $\underline{var}_i$ at the time the file was printed. If $\underline{var}_i$ is a list of the form (var expr) var is written so it will be set to expr, which evaluates at load-time.

  eg If we always want to initialize the variable nr to 0 and stopvar to STOP we do

  ```
  (VARS (NR 0) (STOPVAR 'STOP))
  ```

If the atom :: follows the command, the form following the :: is evaluated and its value is used when executing the command.

eg Suppose we have three lists of functions, fnlist1, fns2 and oldfns, and want all functions on these lists to be printed we can do

```
(FNS :: (APPEND FNLIST1 FNS2 OLDFNS))
```

There are more commands which are described in the LISP manual.

The second argument to makefile is an option list and FAST indicates that the file is printed by print. Without FAST the file is printed by prettyprint. To save space on disc and time use FAST.

20.4 Save. There is also a way to store a users all areas called save on disc store by the function save. He can then start a new LISP run by using this save. This means that we can save the actual status of a LISP run and then start from that point again. In an OS/360 environment we need a sequential dataset for this. In the Appendix the current JCL for doing this is described. The function save gives as value the number of pages saved.

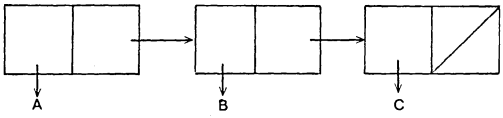
Exercises

Test makefile and load. When you understand them start using them and you will find INTERLISP more easy to work with. The save is useful when loading time of files starts being troublesome.

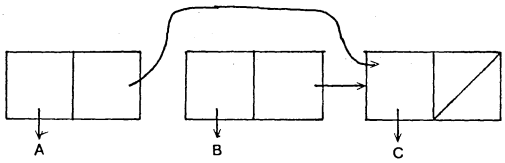
21. Structure-changing functions

21.1 A list cell is created every time the function cons is executed. These cells are allocated from a special list area. There is a maximum number of list cells which can be allocated and for this reason there is a garbage collector which automatically - or by user's call - reclaims all the list cells not longer used and makes these cells available again. List functions - such as append, list etc - use cons, when building lists. This means that the system makes lots of copies of structures. There is however, functions by which we can change an already existing list structure. This section will deal with the functions rplaca, rplacd, nconc, nconc1 and tconc.

eg We can have following structure



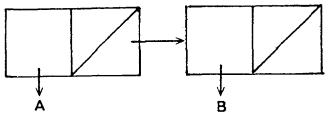
and will change it to



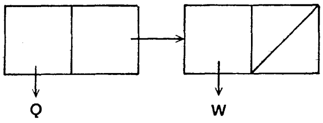
21.2 Functions which changes structure

rplaca[x,y] It replaces car of x to y. If x is an atom, this will set the global value of x to y (see 15.1). If x is a list, the car-pointer of the first list cell of x is changed to y. The value is the changed x.

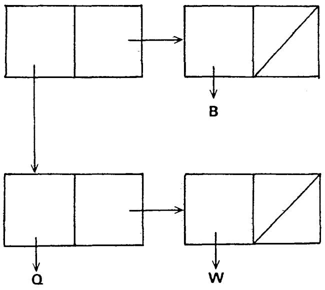
eg X is (A B)



Y is (Q W)



rplaca[x,y] will make the change



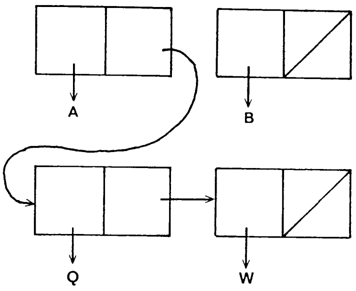
and the value returned is ((Q W) B)

eq[y,car[x]] = T Eq can be used to see if two lists are identical (see Section 3.6)

rplacd[x,y] It replaces cdr of x to y. If x is a list the cdr-pointer of the first list cell is changed to y. We do not use rplacd on atoms, unless absolutely sure. We will then replace x's property-list to y. The system uses property-lists to store internal information which we are not permitted to remove.

eg If x and y had the initial values as in the rplaca example

rplacd[x,y] will make the change



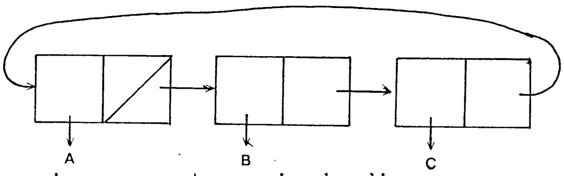
and the value returned is (A Q W)

We can not rplaca or rplacd NIL. An error will then occur.

21.3 By these two functions we can create circular lists.

eg If x is (A B C) then

rplacd[cddr[x], x] is



The print routine can not detect circular lists, so care must be taken that they are not printed.

The following example shows when a circular list can be useful. Suppose we want a function which on succesive calls returns

1, 2, 3, 1, 2, 3, 1, ...

```
(RPAQQ CIRC (1 2 3))
(PROGN (RPLACD (CDDR CIRC) CIRC) 'DONE)
(RPLACA 'CIRCPOINT CIRC)
(DE GENNR NIL (PROG1 (CAR CIRCPOINT) (SETQ CIRCPOINT
                                        (CDR CIRCPOINT>
(GENNR)  gives value 1
(GENNR)  gives value 2       etc ...
```

Notice how progn is used to prevent a circular list from being printed as value and how prog1 is used to get correct value.

21.4 nconc[$x_1,\ldots,x_n$] — $\underline{x_i}$ are lists. Gives value as append (see 11.6), but instead of copying $\underline{x_i}$, the last cdr pointer of each $\underline{x_i}$ is changed. The value is the concatenated list.

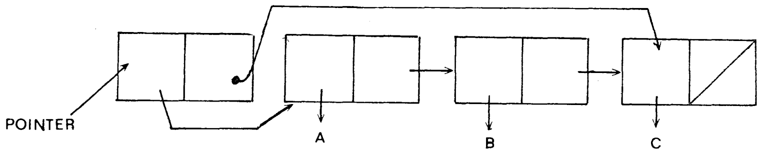
eg nconc[x,y] works as

prog2[rplacd[last[x],y],x]

nconc1[x,y] — Defined as nconc[x,list[y]]. This is as cons but puts y on the end of the list x. The value is x.

tconc[ptr,x] — This function works as nconc1, but instead of always searching to the last element, ptr, is a list cell, where car[ptr] points to the beginning of the list and cdr[ptr] points to the last element. This function is useful when we frequently require to add elements to the end of a list. The value is ptr.

```
eg  (SETQ POINTER (CONS))
    (TCONC POINTER 'A) gives value ((A) A)
    (TCONC POINTER 'B) gives value ((A B) B)
    (TCONC POINTER 'C) gives value ((A B C) C)
```



(CAR POINTER) gives then the list (A B C)

If ptr is NIL, tconc sets up the pointer cell.

```
eg  (SETQ POINT (TCONC NIL 'A))
    (TCONC POINT 'B)
    (TCONC POINT 'C)
    is identical as above.
```

21.5 Some of the list manipulation functions, such as remove, reverse and subst appear also as destructive functions. This means that the original structure is changed. They are then called dremove, dreverse and dsubst.

21.6 Example.

The property-list functions also use these structure-changing functions. Put can be defined as

```
(DE ::PUT (ATM PROP VAL)
  (COND ((NULL (CDR ATM)) (RPLACD ATM (LIST PROP VAL)) VAL)
        ((EQ (CADR ATM) PROP) (RPLACA (CDDR ATM) VAL) VAL)
        (T (::PUT (CDDR ATM) PROP VAL))))
```

Exercises

1-3 Define the functions ::dremove, ::dsubst, and ::dreverse. They work as corresponding functions without the d, but do not use extra list cells.

4. Define the function ::addprop, described in 11.7.

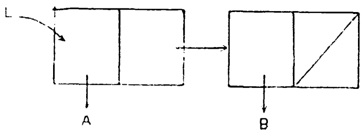
5. There is a system function lconc, which is similar to tconc. By lconc we can concatenate a list at the end instead of only an element. (Compare also nconc and nconc1)

```
eg  (SETQ PTR (CONS))
    (LCONC PTR (LIST 1 2))
    (LCONC PTR (LIST 9 8 7))
    PTR gives then the value ((1 2 9 8 7) 8)
```

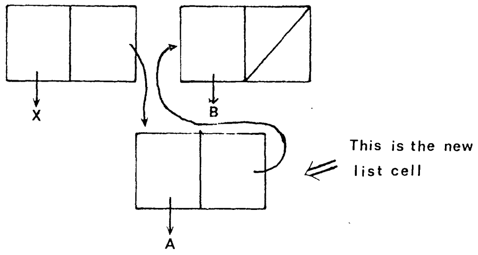
Define ::lconc[ptr,l]

6. Another system function is attach[x,y], which value is the same as cons[x,y], but it attaches x to the front of y by changing (by rplaca and rplacd) the contents of y. The value of attach is eq to y.

eg L = (A B C)



attach[X,1]



This function can be useful if we have several pointers to the same list cell, which can be the beginning of a queue. If we now extend the queue at the beginning, we will still see that all pointers point to the beginning of the queue. If we were doing <u>cons</u> we must reset all pointers ourself.

Define <u>*attach</u>.

7. Let us go back to the <u>tree sort</u> example in Section 9, example 11. In the proposed solution we copy every node in the tree, which is passed when a new node is inserted in the tree. This method consumes a lot of extra list cells. Another solution is instead to merely change one of the pointers in the terminal node to point to the new node. By this method no extra list cells are used. Make necessary changes in the solutions in Section 9, so it uses this new method.

22. Atom and string manipulation functions

22.1 Internally we can create atoms by using the function pack. Pack takes as argument a list of elements and concatenates the pnames of these elements into a new atom.
A pname is the character which will be printed when a LISP expression is printed by prinl.

eg The pname of ATOM is ATOM

The pname of () is NIL

The pname of (A (B)) is (A (B))

The pname of 1.2E+1 is 12.000000

The pname of "STRING" is STRING

eg pack[(A NEW LI SPA TOM)]	= ANEWLISPATOM
pack[(A B (X Y) C)]	= The atom AB(X Y)C, with one blank character
pack[(1 2 3 4)]	= The integer 1234
pack[(1 . 2 E + 2)]	= The floating-point number 120.000000

If the atom packed can be interpreted as a numeric atom, it creates the numeric atom. There is no way to create a "literal number".

22.2 An atom can be taken apart by unpack.

eg unpack[THISATOM] = (T H I S A T O M)

unpack[1234] = (1 2 3 4)

unpack["STRING"] = (S T R I N G)

22.3 Some other useful functions

nchars[x]	number of characters in the pname of x.

nthchar[x,n]	If _n_ positive, it gets the _n_th character in the _pname_ of _x_, and if negative it gives the _n_th character from the end. NIL is returned if _n_ is outside the length of _x_.

```
eg  nchars[ADB] = 3
    nchars[( A () )] = 7, the pname is (A NIL)
    nthchar[LISP,2] = I
    nthchar[LISP,-1] = P
```

gensym[]	Generates a new atom of the form Annnn, where _n_ is digits. A counter is updated at every call, so _gensym_ generates new atoms at every call.

22.4 String functions

stringp[x]	Is _x_ if _x_ is a string, otherwise NIL
strequal[x,y]	Tests if two strings _x_ and _y_ are similar, returns then _x_, otherwise NIL
mkstring[x]	Makes a string of _x_
gnc[x]	_G_et _n_ext _c_haracter of string _x_. Returns the first character and then removes the character. When there are no characters remaining NIL is returned.

```
eg  (RPAQQ STR "STRING")
    (GNC STR)              returns S, an atom
    STR                    "TRING"
    (GNC STR) (GNC STR)
    (GNC STR) (GNC STR)
    STR                    "G"
    (GNC STR)              G
    STR                    ""
    (GNC STR)              NIL
```

glc[x]	_G_et _l_ast _c_haracter. The same as _gnc_, but instead takes the characters from the end.
concat[x_1, x_2, ... ,x_n]	Copies $\underline{x_i}$ and concatenates them to one string

```
eg  concat["A", "NEW", "STRING"] =
                        "ANEWSTRING"
```

substring[x,n,m]	Gets the substring in _x_, from the _n_th character to the _m_th character. Returns NIL if the substring is not well-defined. _n_ and _m_ can be negative with the meaning as _nthchar_ in 22.3 eg substring["A LISP STRING",3,6] = "LISP" substring["A LISP STRING",-6,-3] = "STRI"
mkatom[x]	Makes an atom of the string _x_.
rplstring[x,n,y]	Replaces characters in string _x_, from position _n_ by the characters in string _y_.

In the LISP manual there is a section which describes the internal representation of a string and what happens, when the string functions are applied to different data types.

Exercises

1. Define filename[file], where _file_ is a filename, either with or without generation number, which returns the file name without generation number.

   eg filename[FOO] = FOO

   filename[FOO≠10] = FOO

   For training purposes use two methods. First _unpack_ the atom and pack relevant parts again. Secondly, make a string of the file names and use string functions for finding the relevant part.

2. Write a function strpos[substr,str], where _substr_ and _str_ are strings, one which searches _str_ from the beginning after a sequence of characters equal to _substr_. If a match is found the position of the first character in the sequence is returned otherwise NIL.

   eg strpos["A LISP LIST", "IST"] = 9

   strpos["A LISP LIST","LIS"] = 3

   strpos["A LISP LIST","ALI"] = NIL

23. Arrays

23.1 An array in INTERLISP is a one-dimensional block of storage. An array can be allocated dynamically and is deallocated by the garbage collector. An array is referenced by an array pointer. The elements in an array are referenced by an index, started from 1.

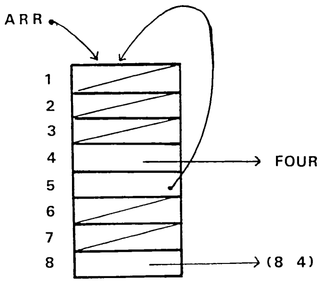
The space in an array can be separated in two sections. The first for storing non-pointer data (an unboxed number) and the second for pointer data. The normal use of arrays are for pointer data. For non-pointer data see the LISP manual. In the pointer section we can store arbitrary INTERLISP pointers, such as atoms, numbers, lists, other arrays etc.

23.2 Functions for handling arrays

array[size,np,initval]	An array of size elements is allocated. The first section will contain np non-pointer data and the second sections will contain size - np pointer data. If np is 0 or NIL the array will only contain pointer data. The elements in the pointer sections are initialized to initval. The value is the array pointer.
arraysize[a]	Returns the size of the array a.
arrayp[a]	Returns a if a is an array pointer, otherwise NIL.
seta[a,n,val]	Gives the nth element of the array a the value val.
elt[a,n]	Returns the value of the nth element of the array a.

23.3 Examples

(SETQ ARR (ARRAY 8))	An array of size 8 is allocated for only pointer data. The elements are initialized to NIL. The variable <u>arr</u> points to the array.
(ELT ARR 2)	Gives NIL as value
(SETA ARR 4 'FOUR)	
(SETA ARR 8 '(8 4))	
(CAR (ELT ARR (PLUS 6 2)))	Gives 8 as value
(SETA ARR 5 ARR)	The fifth element points to the array itself.
(ELT (ELT ARR 5) 4)	Gives the value FOUR.



If we want multi-dimensional arrays we must give the index function ourself. Suppose we want an 8x12 array we can do

```
(DE IND (I J) (IPLUS (ITIMES (SUB1 I) 8) J))
(SETQ ARR2 (ARRAY (ITIMES 8 12)))
(SETA ARR2 (IND 5 11) 'VALUE)
(ELT ARR2 (IND 5 11))
```

Exercises

1. If we want multi-dimensional arrays, we can generate its index function automatically. We can also generate the access function. We introduce two functions <u>defarray</u> and <u>setarray</u> and

they can be described by following an example

(DEFARRAY MAT 3 5 7)	Defines a 3x5x7 array, called _mat_. The function allocates the space for _mat_, generates an index function for it and generates an access function _mat_.
(SETARRAY (MAT 2 4 6) 'VAL)	Gives mat[2,4,6] the value _val_. The generated index function has been used.
(SETARRAY (MAT (IPLUS 2 1) 1 7) 315)	Gives mat[3,1,7] the value 315.
(MAT 2 4 6)	Returns the value of mat[2,4,6], which is _val_. _Mat_ was generated by _defarray_.

Define the functions _defarray_ and _setarray_. In the example, _mat_ is used as the variable, which value is the array pointer and will not be evaluated. This means that the two functions must be of _noeval_ type, but the indices and the value in _setarray_ shall be evaluated.

24. Edit

24.1 The edit was introduced in Section 10 and a number of edit commands were given. The edit contains much more powerful commands and we will describe some of them in this Section, mainly to give an idea as to what can be done by the edit. For full descriptions of the commands a study of the LISP manual is necessary.

24.2 We introduced earlier the current expression, cexp, which is the substructure in the expression we are editing to which our attention is centered. The edit saves the chain of cexp's so we have the possibility to go back to an old one (by the 0 command). Changes done can also be undone by UNDO or for all changes by |UNDO.

24.3 The F (Find) command. By this command we can search for a specified expression. The expression is given as a pattern. The simplest patterns are those where the expression is given explicitly, as in

```
F COND  or  F (SETQ X 10)
```

Elements in an expression can be given implicitly by & and --. & describes an arbitrary element and -- describes a segment (zero or more elements following each other). The expression

```
(SETQ Z (ADD1 X))
```

is matched by following patterns

```
(SETQ & &)
(SETQ --)
(& & &)
(SETQ & (ADD1 &))
```

so we can write

```
F (SETQ & (ADD1 &))
```

The search algorithm for the F command is as follows

a. search on top level on <u>cexp</u>.

b. if not found, search <u>cexp</u> in print order.

c. if not found, the search will ascend to the next higher expression.

Step a. is useful in <u>prog</u>. Suppose we have

```
(PROG (L) (COND ((NULL X) (GO LAB))) ... LAB (SETQ X L) ...)
```

and want to come to the label LAB we simply do

```
F LAB
```

but if we do

```
0 F LAB
```

we go to a higher expression and find then LAB in <u>go</u>, because the label LAB is not on the top level any more. If we have the above example again and first do

```
F X
```

<u>cexp</u> is set to

```
... X)
```

and then again

```
F X
```

no more X is found on <u>cexp</u>, so the search continues on higher levels and <u>cexp</u> is set to

```
... X L)
```

There is also a command BF (Backward Find), which works as find but searches in reverse print order.

24.4 In many situations we want to make changes before or after current expression or to replace it. In order to do this we must come above <u>cexp</u> (by using UP) and then the command (n e_1 ... e_n). There are however, some very convenient commands for this.

NX	Sets <u>cexp</u> to next expression after current expression.
BK	Sets <u>cexp</u> to expression before current expression.
(B e_1 ... e_n)	Inserts e_1 to e_n before current expression.
(A e_1 ... e_n)	Inserts e_1 to e_n after current expression.
(: e_1 ... e_n)	Replaces current expression by e_1 to e_n.
DELETE	Deletes current expression.

24.5 LC location specification. By specifying a position in an expression a more general method can be used. A location specification is a list of edit commands that are executed in a normal way with two exceptions

a. commands not recognized by the editor are interpreted as though they were preceded by F.

eg if cexp is

```
(PROG (X) (COND ((NULL L) NIL) ((NULL A) (SETQ L B))
                      ... >
```

the location specification

```
(LC COND 3 L)
```

specifies the position

```
... L B)
```

b. if a command in a list will cause an error (ie no match can be done by an F command) the editor starts again from the beginning in the command list and goes on searching.

eg If cexp is

```
(PROG (X Y) (COND <L (COND ((CDR L) (SETQ X (CADR L))
                                     (SETQ Y (CAR L)))
                               (T (SETQ X (CAR L>
                     (T (SETQ L (CONS NIL NIL)))) ... >
```

the location specification

```
(LC COND 2 3)
```

will first find the outer cond's first clause but 3 will generate an error, because the clause only contains two elements. Next cond is found and the rest of the commands will fit and we have found

```
(SETQ Y (CAR L))
```

24.6 The A, B and : command are extended also to contain a location specification and we can write

(INSERT e_1 ... e_n BEFORE c_1 ... c_m)

eg (INSERT (CAR L) (GO LOP) BEFORE FOO 3 2)

(INSERT e_1 ... e_n AFTER c_1 ... c_m)

(INSERT e_1 ... e_n FOR c_1 ... c_m)

(REPLACE c_1 ... c_m WITH e_1 ... e_n)

(CHANGE c_1 ... c_m TO e_1 ... e_n)

(DELETE c_1 ... c_m)

$\underline{e}_i$ is an arbitrary LISP expression and $\underline{c}_i$ is an editor command as for the location specification.

Occasionally we want to copy an expression in one place to another. This can be done by the above commands if $\underline{e}_i$ is specified by the ## command.

## specifies a location, but does not change the current <u>cexp</u>. We write

```
(## F COND 2 1)
```

Suppose <u>cexp</u> is

```
(PROG (X Y) (COND ((NULL L)  T) ... ) ... (RETURN 'OK>
```

we can do

```
(REPLACE  T  WITH (## -1))
```

and a copy of the last element in the <u>prog</u> will replace the T in the <u>cond</u>, and we get

```
(PROG (X Y) (COND ((NULL L) (RETURN 'OK)) ... ) ... (RETURN 'OK>
```

24.7 The MOVE command allows us to specify an expression to move, specify the place to move to and specify the operation to be performed at the new place. It looks like

(MOVE c_1 ... c_m TO com c_{11} ... c_{1m})

where <u>c</u> are editor commands as in 24.6 and <u>com</u> is BEFORE, AFTER, : (delete) or list commands as N etc.

We have the following expression as <u>cexp</u>

```
(LAMBDA (X) (PROG (L) (GO LOP) LOP (COND ((NULL L) (RETURN)))
                                                       ... ))
```

If we do

```
(MOVE 3 3 TO AFTER -1)
```

we move

```
(GO LOP)
```

to the end of the <u>prog</u>. We can then do

```
(MOVE 2 1 TO : NULL 2)
```

and we replace <u>l</u> in <u>null</u> to the <u>lambda</u>-variable, and we get

```
(LAMBDA NIL (PROG (L) LOP (COND ((NULL X) (RETURN))) ... (GO LOP)))
```

If several contigous elements will be moved, ie a segment, they can be described by THRU or TO, as in the following examples.

If cexp is

```
(A B C (D E) F (G H) I J K)
```

we can do

```
(MOVE (2 THRU 4) TO BEFORE 7)
```

and we get

```
(A F (G H) B C (D E) I J K)
```

or an identical command is

```
(MOVE (B TO) F) TO BEFORE 7)
```

TO is as THRU except last element is not included.

24.8 Extract and embed. Extraction involves replacing the current expression with one of its subexpressions from any depth and embedding involves replacing the current expression with a subexpression containing it as a subexpression. We have

(XTR c_1 ... c_m)
(MBD e_1 ... e_n)

Suppose cexp is

```
(COND ((NULL L) NIL) (T (PRINT L)))
```

and we want to replace cexp only by

```
(PRINT L)
```

we do

```
(XTR 3 2), (XTR (PRINT L)) or (XTR PRINT)
```

In MDB the current expression will replace every occurence of the atom :: in e_i. If cexp is

```
(PRINT X)
```

and we want to replace it with

```
(COND ((NULL L) (PRINT X) NIL) (T (PRINT X) (GO LOP)))
```

we do

```
(MBD (COND ((NULL L) :: NIL) (T :: (GO LOP))))
```

24.9 Commands that evaluate

E form (E form)	form evaluates and its value will be printed.
(I c x_1 ... x_n)	evaluates x_i and performs then the editor command (C eval[x_1] ... eval[x_n])

(COMS x_1 ... x_n) Each $\underline{x_i}$ is evaluated and its value is executed as an editor command.

24.10 Editor macros

(M macro c_1 ... c_n) Defines macro, which does $\underline{c_1}$ to $\underline{c_n}$, when called.

(M (macro) (arg_1 ... arg_k) c_1 ... c_n) defines macro with arguments.

```
eg  (M (SS) (ELEM) F ELEM 0 P)
    If cexp is
     (A (B C) D)
    and we do
     (SS C)
    the second sublist is found and printed.
```

25. Error handing

25.1 During the development of a program different types of errors will occur. In an interactive environment the system can interrogate with the user and let him decide what to do about the error. In INTERLISP this is done by the _break_ facility, which was introduced in Section 10. In this section we will briefly describe what kind of errors there are and how they are handled by the system. The break facility is described in more detail in Section 26.

25.2 We can distinguish between the following error types:

- unbound atom and undefined function
- illegal arguments to system functions
- user-initiated errors
- other errors, including bugs in the INTERLISP/360-370 system.

25.3 For unbound atom (U.B.A) and undefined function (U.D.F) the interpreter (the _eval_ function) will call the function _faulteval_ and give it the form which caused the error. The form is an atom if unbound atom, or a list if undefined function, with _car_ of the list as the undefined function. _Faulteval_ prints a message U.B.A or U.D.F and calls _break1_. A decision is made here if we should enter the _break_ or if we should return to the top level again. If the error occurred deep in the evaluation a _break_ is made. This is done not to let trivial type-in errors to cause a _break_.

If _faulteval_ returns a value back to the interpreter this value is used exactly as though it was the value from the form. From _break_ a value can be returned by the RETURN command.

In batch, _break1_ will print a backtrace of forms under evaluation, functions entered and variables and its values.

Break commands are described in Sections 10.3, 26.9 and 27.11.

25.4 If we call a system function with illegal arguments an error message is printed and break1 is called, and behaves as described in Section 25.3.

In the LISP manual there is a table with all error messages where messages of the following types can be found,

ILLEGAL RETURN	return not in prog
ATTEMPT TO RPLAC NIL	not allowed to rplaca(d) NIL
NON-NUMERIC ARG	illegal argument to a numeric function
FILE NOT OPEN	read or print to a file not yet opened

25.5 If a pointer references an object outside the range of the virtual address-space a message

REFERENCE OUTSIDE VIRTUAL CORE

is printed. This error can occur if we do car or cdr of numbers for example. Normally this is a user error.

25.6 The user can call the error routines by using the following function

error[mess1,mess1,nobreak]	If mess1 is an atom, mess1 and mess2 are printed on the same line, otherwise a carriage return is made after mess1. If nobreak is T errorb is called otherwise errorx (which calls break1 and we can enter the break).

Example

```
(DE FOO (L) (COND ((NLISTP L) (ERROR L '"IS NOT A LIST" T))
                  (T ..... )))
```

If we call

```
(FOO 'ADAM)
```

the following is printed

```
ADAM IS NOT A LIST
```

and we are back to the toploop again.

With the third argument to error set to NIL the break will be entered and we can get the possibility of correcting the illegal argument l.

25.7 In Section 19.6 we introduced a problem. We defined our own toploop, where we read expressions and gave them to eval. What happens now if an error occurs under the evaluation? In the solution given in that section we are coming back to the system's read-eval-print loop again. This problem can be solved if we use the errorset feature. Instead of calling eval we call the function errorset, which works as eval but catches a return from the error routines.

errorset[u,v]	performs eval[u]. If no error occured under evaluation the value from errorset is a list of the value from eval. If an error occured the value is NIL. The printing of error messages is controlled by v. They are printed if v=T, otherwise not. Notice that errorset first evaluates its arguments and then gives it to eval.

Example. Let us look at a better solution to the example in Section 19.8, where we defined the function applyloop.

```
(DE APPLYLOOP NIL
   (PROG (VAL)
        LOP
        (SETQ VAL (ERRORSET '(APPLY (READ T)
                                    (READ T))
                             T)
        (COND ((NULL VAL) (* AN ERROR HAS OCCURRED)
                                      (GO LOP))
             (T (PRINT (CAR VAL) T)))
        (GO LOP)))
```

errorb[][1]	Returns directly to the last errorset or if no errorset has been done directly to the toploop.[2]
reset[]	Returns directly to the system's toploop.

[1] Is pronounced "errorbang"

[2] An attention-E generates an immediate errorb.

Exercises

1. Many problems, especially combinatorial problems, may often be simply written by using non-deterministic algorithms[1]. The implementation of these algorithms is normally done by backtracking. In a non-deterministic algorithm we can besides the normal statements use a choice statement and statements to report failure and success of the computation. The form of the choice statement can be illustrated by the example

   (CHOICE I (1 2 3 4 5 6 7 8) form)

   which will be interpreted as

   "assign to the variable I a value from the set (1 2 ... 8)

   and evaluate the form"

   The failure and success statements appear

   (FAILURE) and (SUCCESS form)

   where the failure statement works as

   "backtrack to the last choice statement and make a new assignment to the variable and execute the form again, if all values have been taken execute a failure".

   and the success statement is a normal return with form as value.

   Implement these three functions in LISP by using errorset and errorb. Solve then the 8-queen problem by using these non-deterministic primitives.

[1] Non-deterministic algorithms, backtracking and the 8-queens problem can be found in,

Robert Floyd, Non-deterministic Algorithms, JACM vol 14, nr 4, Oct 1967.

26. Break and advise

26.1 In the previous section was described how the break was invoked when an error occurred. This section will describe how the user can make use of the break facility in his program development. This is useful when we want to stop at a specified point in the program and look around in the evaluation environment (stacks, variable bindings etc).

26.2 When we break a function we call that function broken. We can break compiled and machine-coded functions. Actually what happens is that the definition of the broken function is modified. Let us follow an example and see what happens in some different situations

```
-(DE FOO (X) (CONS X X))
FOO
-(BREAK FOO)
(FOO)
```

The function foo is now broken and the next time it will be called the break-loop is entered. We have now the possibility of interrogating the break by using break commands (see 10.3, 26.9 and 27.11) or by giving forms, which will be evaluated.

```
-(FOO (CONS 'A 'B))

(FOO BROKEN)
:BTV
   X (A . B)
FOO
EVAL
   LISPXX (FOO (CONS (QUOTE A) (QUOTE B)))
LISPX
   .
   .
   .
:GO
FOO = ((A . B) A . B)
((A . B) A . B)
```

The break command BTV prints a backtrace of functions and variables bound on the stack, and the break is released by the command GO. The computation continues and the result is printed.

The system knows about all broken functions thus makefile and prettyprint will work on the original function definition even if the function is broken.

```
-(PP FOO)
<FOO
    (LAMBDA (X)
       (CONS X X>
-(PRINTDEF (GETD 'FOO))
<LAMBDA (X)
   (BREAK1 (PROGN (CONS X X)) T FOO NIL>
```

In the function cell a modified version of foo is placed, but when pp is used for a prettyprint the original version of foo is used.

We can now unbreak foo by

```
-(UNBREAK FOO)
```

We can also give a condition when the break shall occur, such as

```
-(BREAK (FOO (ATOM X)))
(FOO)
-(FOO '(A B C))
((A B C) A B C)
-(FOO 'A)
(FOO BROKEN)
:X
A
:(SETQ X (LIST X))
(A)
:OK
FOO
((A) A)
```

The break will only appear when foo is called with an atom as argument. In the example we look at the value of x, (x is no break command and is evaluated as a form) and then rebound it to the list of x. The break is released by OK (the value from the computation of foo is not printed).

When defining the break a list of break commands can be given directly. Instead of the break-loop the elements of that list are treated as break commands. Trace is implemented by this feature.

```
-(BREAK (FOO (ATOM X) ((PRIN1 '"X IN FOO IS ") (PRINT X) GO)))

(FOO)
-(FOO 'A)


(FOO BROKEN)
X IN FOO IS A
FOO = (A . A)
(A . A)
```

If a function is called from many different places, but our interest is only to break the function when it is called from a specified function we do

```
-(BREAK (CONS IN FOO))
(CONS-IN-FOO)
```

Cons will be broken only when called inside foo. The call to cons in foo will be changed to a call to the function cons-in-foo, which instead will be broken.

```
-(PRINTDEF (GETD 'FOO))
<LAMBDA (X)
   (CONS-IN-FOO X X>
-(PRINTDEF (GETD 'CONS-IN-FOO))
   <LAMBDA (U V)
   (BREAK1 (PROGN (CONS U U))
                  T CONS-IN-FOO NIL>
```

The information of a break is saved on the broken functions property-list. This information can be used if we later want to make the same break again by simply doing

```
-(REBREAK FOO)
```

26.3 In a prog interest is usually in breaking a function at a label or at a special statement and this can be done by the function breakin. An example illustrates its use

```
-(DE FOO (X)
-    (PROG (L)
-         LOP
-         (COND ((ZEROP X) (RETURN L)))
-         (SETQ L (CONS X L))
-         (SETQ X (SUB1 X))
-         (GO LOP>
FOO
-(BREAKIN FOO (AFTER LOP))
FOO
-(FOO 3)
                                  Enter the break after the label lop.
((FOO_G) BROKEN)
:L
NIL
:OK                               Go on
(FOO_G)                           Leave the break


(FOO_G) BROKEN)                   New break at the same place as before
:L                                Look at l
(3)
:(SETQ X 0)                       Rebound x
0
:OK                               Go on
(FOO_G)                           Leave the break
(3)                               Value from (FOO 3)
-(UNBREAK FOO)
(FOO)
-(BREAKIN FOO (AROUND (SUB1 X)) (LESSP X 7)
          (RETURN (PLUS X -2)))
(FOO_G)
```

If the condition that x is less than 2 is satisfactory and the break occur just before the evaluation of sub1 the last argument is a break-command list as described before and it will leave the break by the value of plus[x,-2]. This means that the sub1[x] will not be evaluated and the value from the RETURN command is taken as the value instead.

```
-(FOO 8)

((FOO_G) BROKEN)
(FOO_G) = 4
((FOO_G) BROKEN)
(FOO_G) = 2

((FOO_G) BROKEN)
(FOO_G) = 0
(2 4 6 7 8)
```

FOO_G is an internal description that foo is broken inside somewhere.

26.4 Notice that the editor will work on the broken function. If you want to edit your original function you must first unbreak it and after the editing rebreak it.

26.5 Break functions.

break0[fn,when,coms]	sets up the break by redefining fn . Break and trace will call break0.
break1[brkexp,when,fn,coms]	calls the break. If when evaluates to NIL brkexp is evaluated and returned as the value of break1. If when evaluates to true a break will occur. If coms is NIL then the break-loop is entered, otherwise the break commands are taken from coms.
break[$x_1,\dots,x_n$]	a nospread nlambda function. Each x_i describes a function to break. x_i can either be a function or a list (fn when coms).
trace[$x_1,\dots,x_n$]	as break but traces the functions instead.
unbreak[$x_1,\dots,x_n$]	If x_1 is NIL all broken functions are unbroken, and all information saved are thrown away. If x_1 is T the latest broken function is unbroken, otherwise each x_i describes a function, which is unbroken.
rebreak[$x_1, \dots, x_n$]	Every function x_i will be broken again exactly as i was previously broken without having to re-specify all the break information again. If x_1 is NIL or T rebreak works as unbreak.

fn can either be a function name or a list (fn1 IN fn2). when is a form which determines if a break shall occur or not.

coms is a list of break commands, or forms to be evaluated. For commands which are inputed as a number of expressions, ie

```
?= A B
```

they are written as

```
( .... ? (A B) ..... )
```

If we want the value from a form printed we must print it ourselves.

```
((PRINT X) .... )
```

We can introduce breakmacros by extending the variable breakmacros by a list of the form (macro command1 ... commandn). The atom macro can then be used in coms as a break command.

26.6 Examples

If we do

```
(TRACE FOO)
```

trace sets up the following command list

```
(TRACE ?= NIL GO)
```

where trace is a special flag indicating that the message

```
"function"
```

is printed.

?= is a break command which prints the variable and values for the broken function.

The break is set up by a call to break0 as

```
(BREAK0 'FOO T '(TRACE ?= NIL GO))
```

Foo is now redefined and contains a call to break1

```
<FOO
(LAMBDA (X)
   (BREAK1 (PROGN (CONS X X)) T FOO (TRACE ?= NIL GO>
```

When foo is called and the printout is made the break is released by GO, which evaluates the first argument to break1, which is the original definition of foo; and prints its value.

26.7 breakin[fn,where,when,coms] fn, when and coms are as in break. Where specifies the location where the break is inserted. The location is specified as a list started by either BEFORE, AFTER or AROUND followed by editor commands specifying the location (see 24.5) ex (BEFORE COND)

specifies the point before the first occurence of cond.

(AFTER 3 2)

specifies a point after the second sublist in the function body; in

(LAMBDA (X Y) (PROG (L N) (SETQ X L) ..)

it is before the setq.

(AROUND (SETQ X Y))

specifies that the break occurs just before the evaluation of the expression mentioned. In this case the user can use the EVAL command for evaluating the expression and look at its value afterwards. The variable /VALUE is bound to the value.

Multiple break points can be inserted. Breakin can only be used in interpreted functions.

26.8 The function virginfn[fn] gets, regardless of any amount of breaks, breaking etc, the original version of your function fn.

26.9 We have introduced additional break commands and here is a summary of them.

GO	Releases the break and allows the computation to proceed. The brkexp is evaluated and this value is printed. See earlier how GO is used by the trace and where brkexp is the original function definition.
OK	Same as GO except that the value is not printed.
EVAL	Same as GO and OK but the break is maintained after the break. The variable /VALUE contains the value from brkexp and can be examined. If GO or OK follows EVAL the brkexp expression will not re-evaluate brkexp.
RETURN form	Releases the break and form is evaluated as brkexp. This command is normally used when the break occurred depending on an error and the value from form is taken as the value from the erroneous expression, which caused the break.

?= Will print the variables and its values for the broken function, (or to a function pointed to by <u>lastpos</u>, (see next section). For an extended use see the LISP manual.

Break commands are described in 10.3, this section and in 27.11. The command ? gives a list of available break commands.

26.10 <u>Advising</u>. By advising we can change the interface between functions. This means that we can modify a function by placing new code before or after the computation of the function. Examples of this use are <u>break</u> and <u>trace</u>, which modifies the function by putting code so it calls the break-package. By advising it is not necessary for the user to know how the function works, he can modify them without concern for their contents and details of operations. Advising works as <u>break</u> on machine coded, compiled and interpreted functions, and it is possible also to advise a function only when called from some other specified functions.

26.11 If we have the following definition

```
(LAMBDA args body)
```

the corresponding advised function is

```
(LAMBDA args
   (PROG (/VALUE)
         (SETQ /VALUE (PROG NIL
                            advise1
                              .
                              .        ADVISE BEFORE
                              .
                              .
                            advisen
                            (RETURN body)))
         advise1
           .
           .        ADVISE AFTER
           .
           .
         advisen
         (RETURN /VALUE)))
```

26.12 Advise functions.

advise[fn,when,where,what] Advise the function _fn_, _when_ = BEFORE or AFTER, _where_ specifies where among the advises this new advice is put, can be specified as LAST (NIL) or FIRST or by editor commands, _what_ specifies the code to put in.

```
eg  (ADVISE 'FOO 'BEFORE 'LAST '(SETQ X NIL))
    (ADVISE '(CAR IN FOO) 'AFTER NIL '(PRINT /VALUE))
```

unadvise[x_1 ... x_n] as _unbreak_.

readvise[x_1 ... x_n] as _rebreak_, information is saved about earlier advises on $\underline{x_i}$.

26.13 _Example_. Suppose we want to have statistics about how many times particular functions are called under a computation. By _advise_ this is simple. First we define a function _stat_, which sets up the advise.

```
(DF STAT L
   (MAPC L (FUNCTION (LAMBDA (FN)
                          (PROG (STATVAR)
                                (SETQ STATVAR (PACK (LIST FN '=STAT)))
                                (SET STATVAR 0)
                                (ADVISE FN 'BEFORE NIL
                                    (SUBST STATVAR
                                           '::VAR::
                                           '(SETQ ::VAR:: (ADD1 ::VAR::))))
                                (SETQ STATFNS (CONS (CONS FN STATVAR)
                                                    (STATFNS))
                                (RETURN>
```

If we now do

   (STAT FOO FIE)

we create the global variables, as counters

   FOO=STAT and FIE=STAT

and initialize them to 0.
For _foo_ the advise-expression is

   (SETQ FOO=STAT (ADD1 FOO=STAT))

and we save the function and the counter on an association list for later use when the report of statistics is printed.

<u>Exercises</u>

1. Continue the work in the example in 26.13. Define a function <u>printstat</u>, which prints a table of functions and its frequencies. Define also a function <u>unstat</u> and <u>restat</u>, which removes resp initializes the statistics.

   Can we also use the functions with

   ```
   (CAR IN FOO)
   ```

   as argument?

27. Stack functions

27.1 During the evaluation of an expression the interpreter uses stacks for saving information. The control-stack contains function returns so that the system knows in what order functions have been called. The parameter-stack contains variable names and values. Temporary results etc are also saved on stacks. It is more convenient to consider the stacks as one. This single stack will then contain function blocks of all functions that have been entered but as yet not exited. A function block consists of the function name, variable names and values. We will in this section describe some functions by which we can use to gain access to the stacks.

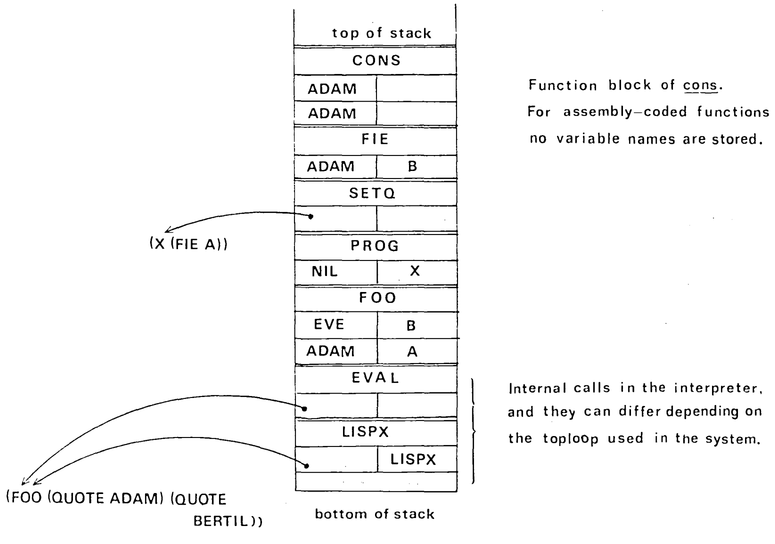
27.2 Let us follow an example where foo and fie are defined as

```
(DE FOO (A B) (PROG (X) (SETQ X (FIE A)) ... )
(DE FIE (B) (CONS B B))
```

If we evaluate

```
(FOO 'ADAM 'EVE)
```

the stack will contain the following information at the moment when the cons in fie is evaluated.



top of stack
CONS
ADAM
ADAM
FIE
ADAM
B
SETQ
(X (FIE A))
PROG
NIL
X
FOO
EVE
B
ADAM
A
EVAL
LISPX
LISPX
(FOO (QUOTE ADAM) (QUOTE BERTIL))
bottom of stack
Function block of cons. For assembly-coded functions no variable names are stored.
Internal calls in the interpreter, and they can differ depending on the toploop used in the system.

27.3 The compiled functions will also put its names of variables on the stack although they are not used. This is done for compatability with interpreted functions. In a compiled function the values are picked up from known positions in the stack instead of doing a search. This scheme is used to let free variables be used between compiled and interpreted code. It is also very useful as symbolic debugging information and is used by the backtrace.

27.4 A position in the stack can either be the beginning of a function block (actually a position in the control stack) or to a variable-value pair (a position in the parameter stack, called slot). A stack position is a datatype in INTERLISP and is referenced by a pointer in the same way as an atom, list, array etc.

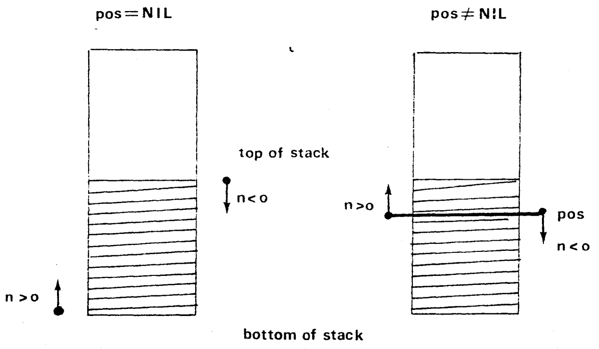
27.5 Stack functions for accessing a function block.

stkpos[fn,n,pos]	Returns the stack position for the nth function block of fn starting at position pos. If n is positive the stack is searched from the bottom and if n is negative the stack is searched from the top. If n is NIL, -1 is used. If pos is given the search starts at that position.
stknth[n,pos]	Returns the stack position for the nth function block relative to position pos. If pos is NIL the bottom of the stack is assumed if n>0 and the top of the stack if n<0.
stkname[pos]	Returns the name of the function in the block of position pos.

To ge the top of the stack (current position) do stkpos[] .
In stkpos and stknth the position pos can be given as a literal atom and is then treated as the position

```
(STKPOS pos -1)
```

27.6 To clarify how the search is done study the two figures



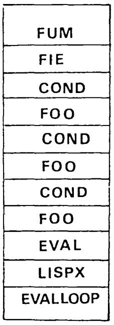
27.7 Example. Suppose we have foo and fie defined as

```
(DE FOO (N M) (COND ((ZEROP N) (FIE M))
                    (T (FOO (SUB1 N) (PLUS N M>
(DE FIE (I) (ADD1 (FUM (SUB1 I>
```

Fum is defined, so it will print the result from a number of examples, where the stack functions are used. Suppose we evaluate

```
(FOO 2 5)
```

the stack will have the following status at the moment fum is called



The variables are not shown, only the order between the function blocks

```
(DE FUM (N)
   (PRINT (STKPOS))                   #2600A164, stack position to fum
   (PRINT (STKPOS 'FIE))              #2600A12C, stack position to fie
   (PRINT (STKNAME (STKPOS)))         FUM
   (PRINT (STKNAME (STKPOS 'FOO 3)))  FOO, the 4th block from the top
   (PRINT (STKNAME (STKNTH 4)))       FOO, the 8th block from the top
   (PRINT (STKNAME (STKNTH -10)))     LISPX
   (PRINT (STKNAME (STKNTH 2
             (STKPOS 'EVAL))))        COND, the 7th block from top
   (PRINT (STKNAME (STKNTH 2 'COND)))  FUM
   (PRINT (STRPOS 'COND -1 'FOO)))) COND, the 5th block from top
```

27.8 Stack functions for accessing information in a function block.

stknargs[pos]	Value is the number of arguments bound by the function at position pos.
stkarg[n,pos]	Value is the slot for the nth argument of the function at position pos.

27.9 When the stack position is a variable-value pair, as from stkarg, we can get the variable name by making cdr of the position and the value by car. By rplacd and rplaca the variable name and value can be changed.

27.10 Example. We want a function mkass[pos], which returns an association list of the variables and values bound in the function block at position pos. With the stack as in 27.7.

```
   (MKASS (STKPOS 'FOO 1))  returns ((N . 2) (M . 5))
   (DE MKASS (POS)
      (PROG (NARGS AL TEMP)
            (SETQ NARGS (STKNARGS POS))
            LOP
            (COND ((ZEROP NARGS) (RETURN AL)))
            (SETQ TEMP (STKARG NARGS POS))
            (SETQ AL (CONS (CONS (CDR TEMP) (CAR TEMP)) AL))
            (SETQ NARGS (SUB1 NARGS))
            (GO LOP)))
```

27.11 When we enter a break (in interactive mode) we can see the stack by the commands BT, BTV, BT* and BTV/. There is also a function baktrace, which can be used for printing the information on the stack.

BT prints the functions.

BTV prints functions, variables and values.

BTV* prints as BTV and forms under evaluation.

BTV/ prints everything on the stack.

There is a possibility to work with stack -positions in break with the @-command. By doing

```
   @ FOO
```

we set lastpos to the function block for the last call to foo. (the first function block for foo from the top of the stack). Some break commands are effected by lastpos and among them are ?=, BT, BTV, etc. By doing

```
@3
```

we move lastpos three function blocks down the stack and by

```
@-3
```

we move lastpos three blocks up.

27.12 When the interpreter looks up a value of a variable, the parameter stack is search from the top after the first occurrence of a slot, containing the variable name. Following two functions can force the search to start from an arbitrary position in the stack.

stkscan[var,pos]	Start the search at position pos, returns a pointer to the slot, if var is stored on the stack, otherwise it returns var.
stkeval[pos,form]	Form is evaluated in such a way that all variables are searched as stkscan.

Example. We want a function mapev[l], where l is a list of forms. Mapev shall evaluate every form and return a list of the computed values.

```
(SETQ L '(A B C D))
(MAPEV '((CONS 'Q (CDR L)) (CAR L))) = ((Q B  C D) A)
```

Let us first see what happens if we define mapev as

```
(DE MAPEV (L)
   (COND ((NULL L) NIL)
         (T (CONS (EVAL (CAR L)) (MAPEV (CDR L))))))
```

If we use the above example the result will be the erroneous

```
((Q (CAR L)) (CAR L))
```

Why? When we called mapev, the variable l in the forms was expected to have its global value, but mapev has l as lambda-variable and has therefore been bound on the parameter stack. What we wanted to do was to evaluate every form in the environment before mapev was called the first time. This can be done if we instead define mapev as

```
(DE MAPEV (L)
   (COND ((NULL L) NIL)
         (T (CONS (STKEVAL (STKNTH -1 (STKPOS 'MAPEV 1))
                                    (CAR L))
               (MAPEV (CDR L))))))
```

Observe that the stack position is to the function block before the first call to mapev - mapev is called recursively - We don't allow the forms to contain new calls to mapev or other stack functions which can change the stack.

27.13 There are two functions by which we can clear the stack and return directly to a function, which has been entered but not yet exited.

retfrom[pos,val]	will clear the stack down to the function block at position pos, and return from that function with the value val.
reteval[pos,eval]	works as retfrom, but will evaluate form after the return to the function block at pos and return that value.

Example. Suppose we have a function, which normally goes very deep into the recursion, finds a value and returns the same value in every step up to the top level. Instead of doing the normal returns we can use retfrom. A simple example of such function is to find the last element on a list.

```
(DE LASTLONG (L)
    (COND ((NULL (CDR L)) (RETFROM (STKPOS 'LASTLONG 1) (CAR L)))
          (T (LASTLONG (CDR L)))))
```

Exercises

1. Write a function ppdump[pos1,pos2], which prints variable names and values from the parameter stack starting at pos1 and down to pos2. A position can be given as a stack position to a function block or as a function name (same meaning as with stkpos). If pos1 is NIL start from the top, if pos2 is NIL end at bottom and if pos2 is a number n, end at the nth function block below pos1.

2. Define the function ::reset, described in 25.7.

28. Funarg

28.1 In the previous section (27.12) we illustrated by an example of what can happen when unexpected collisions of variable names occurred. When we discussed _map_-functions in Section 18 we said that a functional expression shall be "quoted" by _function_ instead of _quote_. This was also to prevent variable collisions. The problem is that in LISP we are allowed to create expressions which later will be evaluated in another environment. If these expressions contain free variables, we sometimes want these free variables to have the value they had at creation time of the expression and sometimes the value they had at evaluation time. By _funarg_ this is solved for functional expressions and in this section is described by some simple examples when the funarg feature is useful.

28.2 Let us first examine _function_ again

function[fn,freevars]	If _freevars_ is NIL then it is identical to _quote_, but helps the compiler to show that this is a functional argument. When _freevars_ is ≠ NIL it is a list of variable presumably free in _fn_. If _freevars_ is an atom it is evaluated and the result is taken as the list of free variables. A _funarg_ expression will then be created.

eg Suppose _foo_ is defined as

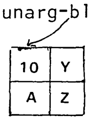
```
(DE FOO (X) (CONS X (CONS Y Z)))
```

in which _y_ and _z_ are free variables. If we perform the following computations

```
(SETQ Y 10)
(SETQ Z 'A)
(SETQ FN (FUNCTION FOO (Y Z)))
```

we will create a funarg-expression

```
(FUNARG FOO funarg-block-pointer)
```



To simplify the notation we can write

```
(FUNARG FOO [(Y . 10) (Z . A)])
```
[1]

The funarg-block works as a mini-stack, on which y and z are bound to their current values. If we reset y and z

```
(SETQ Y 20)
(SETQ Z 'B)
```

and then evaluate

```
(APPLY* FN T)
```

we get the value

```
(T 10 . A)
```

When the funarg-expression was applied its mini-stack was put on top of the parameter stack and when the interpreter searched for the values of y and z they were now bound on the stack and their global values were never reached.

28.3 Funarg is not a function itself, but is recognized by the interpreter in the same way as lambda or nlambda, but only in the context when the funarg-expression is applied to some arguments. In other words, the expression

```
(FUNARG fn-expr funarg-block-pointer)
```

is used exactly like a function.

28.4 Example where variable collisions can occur. In Section 18, example 1 we defined a function calc, so we could write

```
(CALC (FUNCTION PLUS) 10 20)
```

to get value 30, or

[1] In LISP 1.5 the funarg is usually implemented by an association list.

```
(CALC (FUNCTION (LAMBDA (X Y) (COND ((GREATERP X Y) X) (T Y))))
     10 15)
```

to get the maximum value of 10 and 15.

In the solutions calc is defined as

```
(LAMBDA (FN A B) (APPLY* FN A B))
```

Let us see what happens if we define a function foo as

```
(LAMBDA (NR)
   (PROG ((A 5) VAL)
         :
       <SETQ VAL (CALC <FUNCTION (LAMBDA (X Y) (IPLUS X Y A>
                           10 NR>
         :
       ))
```

In the function expression we use the variable a as a free variable and we also use a as a lambda-variable in calc, this will cause some troubles. In the evaluation of the apply* in calc we will evaluate the function expression and look for the value of a. We will now find the value of the lambda-variable in calc, when we meant the prog-variable in foo. This problem can be solved by at least three different solutions

- change the name of the prog-variable a in foo.
- change the name of the lambda-variable a in calc. A good rule is to have funny names in functions with functional arguments, so we could better call them calc*op, calc*a and calc*b.
- let the second argument to function be a list of the free variable a used in the expression, so we get

```
<SETQ VAL (CALC <FUNCTION (LAMBDA (X Y) (IPLUS X Y A)) (A)>
                     10 NR>
```

28.5 Example by a random number generator. Suppose we have a function random[lim], which returns a random number in the interval[1,lim]. The free variable randnr contains a number, which random uses for making calculations on to produce the next random number. Random can be defined as

```
(DE RANDOM (LIM)
   (SETQ RANDNR (ABS (ITIMES RANDNR RANDNR)))
   (ADD1 (IDIFFERENCE RANDNR (ITIMES (IQUOTIENT RANDNR LIM) LIM>
```

randnr is initialized to an appropriate number, ie 12345. If we now want to have two instances of this random number generator in different states and run them independently we can use the funarg feature. To get two instances we can do

```
(SETQ RAND1 (FUNCTION RANDOM (RANDNR)))
(SETQ RAND2 (FUNCTION RANDOM (RANDNR)))
```

A funarg-expression is now created for each of the instances, randnr is bound to its initial value and looks like

```
(FUNARG RANDOM [(RANDNR . 12345)])
```

To use an instance of the generator we do

```
(APPLY:: RAND1 50)  or (APPLY:: RAND2 200)
```

Changes of the free variable randnr in one instance will not effect the other instance neither the global variable randnr.

Actually there is a random number generator, the function rand. It contains also a global variable, ranstate.

Exercises[1]

1. An artificial example of funarg found in the literature.

```
(DE F (X) (COND ((ZEROP A) X)
                (T (MINUS X))))
(DE G (X) (PROG (A)
                (SETQ A 2)
                (RETURN (FUNCTION F (A)))))

 (PROG (A B H)
       (SETQ A 0)
       (SETQ H (G 2))
       (SETQ B (APPLY:: H 3))
       (RETURN B))
```

   What value will be returned from the prog?

   If we change the call to function in g to

```
(FUNCTION F)
```

   what value will then be returned?

2. By funarg we can implement simple processes. The random number generator in 28.5 can be seen as such a process.

   A process can vaguely be defined as a function, which can

[1] These examples are to be found in Sandewall (ref.5).

appear in different instances, where every instance has its own values on some variables. The variable randnr was such a variable. Suppose we want to define a function dp (define process) in analogy with de and df, so that

```
(DP RANDOMPROCESS (RANDNR) (LIM) ++++++)
```

defines randomprocess to be process generator. Evaluating the expression

```
(RANDOMPROCESS 123456)
```

will return as value a process, where the variable randnr is initialized to 123456 and where lim is a parameter given each time the process is called and where ++++++ is the form then to evaluate. Define the function dp.

Define then a process spit, in such a way that if

```
(SPIT '(A B C D E))
```

is evaluated a process is generated which on its first call returns A, then B etc. until E and then NIL on all successive calls.

Define also a process alternate, which has two processes as local variables and which on the first call calls the first process, next call the second process and then the first again etc.

```
(ALTERNATE (SPIT '(A B C D E F))
           (SPIT '(1 2 3 4 5 6 7)))
```

will generate a process, which on successive calls will return

A, 1, B, 2, C, 3, etc

29. Compiler and assembler

29.1 Although LISP is an interpretative language it needs a compiler to produce more efficient code (machine code). As a beginner in LISP you need not worry about the compiler until you have reached the stage of having a large file of well-tested functions and you want your functions in "production". Some of the utility packages, such as break and advise can be used on compiled code, but you can not of course use the editor on such functions. The compiler can be used to compile single functions or a whole file produced by makefile. The assembler can be used by the user if he wants to specify a sequence of machine instructions in the LISP code, otherwise it is used by the compiler in its last pass.

Most of the INTERLISP system is coded in LISP, including the compiler. This code has then been compiled by a bootstrap process. First the LISP-coded-compiler was compiled by itself and then the compiled version of the compiler was used to compile the rest of the system.

29.2 The compiler will (in interactive mode) ask some questions concerning the compilation, and want to know the following:

I. Want to see macro- or machine-code? Usually not of interest to the user.

II. Want to redefine the functions you are compiling? Can be of interest only to output the code to a file, and still working on the non-compiled version of the functions.

III.Want to save the old function definitions on the property-list of the function name? Can be of interest if you still want to be able to edit the functions just being compiled.

IV. Where to save the compiled code? Important.

There are five global variables (lapflg, lstfil, strf, saveflg and lcfil), which will be set depending on the answers.

Question and proper answers are:

I. LISTING?

Answers:

1,2 or YES prints code (for details see LISP manual)

NO no code is printed.

There are other possible answers to this question and each of which specifies a complete compilation.

S Same as last compilation. The global variables are not changed.

F Compiles to a file. No redefining of functions (II) and no definitions are saved (III).

SF Store the new definitions. Redefining of functions are done (II) and old function definitions are saved (III).

II. (STORE AND REDEFINE?)

Answers: YES or NO

III. (SAVE EXPRS?)

Answers: YES or NO

IV. (OUTPUT FILE)

Answer: A file name. If NIL is given no output will be done.

In a batch environment the global variables can be set as answers to the questions.

lapflg and lstfil used by I. They are not initialized to any list code.

strf is set to T if YES on question II, otherwise NIL.

svflg is set to T if YES on question III, otherwise NIL.

lcfil is set to the file name given on question IV.

29.3 The normal use of the compiler is to answer the first question by either F or ST.

Example.

Functions foo and fie are defined and we want them to be compiled, and want to replace the old definitions in core by their compiled version respectively.

```
eg  -(COMPILE '(FAK FACT))
    LISTING?
    ST
    OUTPUT FILE:
    NIL
    (FAK COMPILING)
       .
       .
       .
    (FAK REDEFINED)
    (FACT COMPILING)
          .
          .
          .
    (FACT REDEFINED)
    (FAK FACT)
```

In batch mode we do

```
eg  (SETQ LAPFLG 'ST)
    (SETQ LCFIL NIL)
    (COMPILE '(FAK FACT))
```

29.4 A more convenient way to use the compiler is to let a whole file (a file created by _makefile_) be compiled at the same time. This is done by the two functions _tcompl_ and _recompile_.

Compiler functions

compile[x]	compiles each function on the list x.
tcompl[files]	_files_ is a list of symbolic files, _tcompl_ compiles the functions in every such file and creates a compiled version of the file. The name convention for the compiled file is to prefix the symbolic file name by an _c_.

eg (TCOMPL '(FOO FIE))

creates the files CFOO and CFIE.

recompile[pfile,cfile,fns,prettyfns,prettyvars]

By _recompile_ the user can update a compiled file without recompiling all functions on the file, _pfile_ is the name of the symbolic file to be compiled, _cfile_ the name of the compiled file from which definitions may be copied, _fns_ is a list of functions in _pfile_ to recompile or if _fns_ is T all functions on _pfile_, which now are defined as _expr_'s (which presumably have been edited and therefore changed). If _prettyfns_ and _prettyvars_ (the variables used by _makefile_) are given the _pfile_ is not loaded and functions defined by these variables are taken from the core directly.

tcompl and _recompile_ will ask the standard compiler questions, except for the output file.

The following examples will demonstrate some different uses of these functions.

```
    (SETQ FOOFNS '(FOOA FOOB FOOC))
    (MAKEFILE 'FOO)                      Symbolic file foo created.
    (TCOMPL  '(FOO))                     Compiled file cfoo created.
 ......  eventually new session  ......
    (LOAD 'CFOO)                         Load compiled file cfoo.
    (LOAD 'FOO 'PROP)                    Load symbolic file foo, but
                                         store function definition  on
                                         the property-list instead of
                                         redefining the function.
    (EDITF FOOB)                         Edit function foob, the editor
                                         finds the symbolic version and
 ...  edit commands  ...                 will unsave it after editing.
    OK
    (MAKEFILE 'FOO)                      New symbolic file foo created.
    (RECOMPILE 'FOO 'CFOO '(FOOB))       New compiled file cfoo created,
                                         where foob is recompiled and
                                         fooa and fooc are copied from
                                         old cfoo.
```

Instead of giving the list of functions to redefine explicitly we can give T as in

```
   (RECOMPILE 'FOO 'CFOO T)
```

and all functions that are now defined as _expr_'s are redefined.

If the symbolic file, _pfile_, is in core it is more efficient to give _foofns_ and _foovars_. _pfile_ does not need to be loaded and the contents of the file is given by these two variables. In the above example we could instead do

```
   (RECOMPILE 'FOO 'CFOO T 'FOOFNS 'FOOVARS)
```

or simply

```
   (RECOMPILE 'FOO)
```

29.5 The compiler must know to what type of function belongs. First it looks in the function cell to see if it is defined. If it was not, we must supply the information by including the function name of two different lists

nlama	(for <u>nlam</u>bda <u>a</u>toms)	noeval-nospread functions
nlaml	(for <u>nlam</u>bda <u>l</u>ists)	noeval-spread functions

If a function is not on these lists it is assumed to be an <u>eval</u> function. The global variable <u>alams</u> is set to the functions assumed by the compiler to be of <u>eval</u> type and this list can be examined after the compilation.

29.6 The user can also effect the compilation by introducing <u>compiler macros</u>. One type of macros is <u>open macros</u>, where the call to a function is replaced by the code in the macro definition. This can save compute time because we save function calls, but the code will normally be larger.

Example. Suppose <u>foo</u> is defined as

```
(LAMBDA (L) (FIE (CONS1 L)))
```

and <u>cons1</u> is defined as

```
(LAMBDA (X) (CONS X X))
```

If we compile <u>foo</u> a call to <u>cons1</u> will appear. An open macro for <u>cons1</u> can be defined by

```
(PUT 'CONS1 'MACRO '(LAMBDA (X) (CONS X X)))
```

and <u>foo</u> is then compiled as if it was defined as

```
(LAMBDA (L) (FIE ((LAMBDA (X) (CONS X X)) L)))
```

Another type of compiler macro is <u>substitution macros</u>, where a form is replaced by a macro definition in which, lambda-variables are substituted against corresponding arguments in the form.

Example. If we define a substitution macro for <u>cons1</u> by

```
(PUT 'CONS1 'MACRO '((X) (CONS X X)))
```

<u>foo</u> is compiled as if it was defined as

```
(LAMBDA (L) (FIE (CONS L L)))
```

The third type of macros is <u>computed macros</u>, where the form will be replaced by the expression we get as value when the macro definition is evaluated.

More details about these different types of macros can be found in the LISP manual.

30. Miscellaneous

30.1 date[] Returns the date as a string on the form "dd-mm-yy hh:mm:ss"

clock[n] When n=2 it gives the number of milli seconds of compute (CPU) time since this INTERLISP run was started up. To measure the CPU time for a computation, do clock[2] before and after it and take the difference.

tsop[] Gives T as value if we are in interactive mode and NIL in batch mode.

30.2 reclaim[] Initiates a garbage collection.

Normally the user does not need to worry about garbage collection. It is initiated automatically when the storage assigned for a data type has been exhausted.

30.3 mkn[p] Makes an integer of the pointer p.

unbox[n] Makes a pointer of the integer n.

eg (ADD1 (MKN 'KALLE))

takes the address to atom KALLE, makes a LISP integer of it and adds one to it.

A more useful example is the following. Suppose we want to print all atoms in the system. The atoms are allocated sequential and each atom occupies 16 bytes. The first atom is NIL. We can now define a function printatoms as

```
(DE PRINTATOMS NIL
   (PROG (ADR ENDADR)
         (⁚ MAKE A NUMBER OF THE ADDRESS TO THE FIRST ATOM NIL)
         (SETQ ADR (MKN  NIL))
         (⁚ TERMINATE THE PRINTING OF THE ATOMS WHEN THE ATOM
            PRINTATOMS IS REACHED, SO MAKE A NUMBER OF ITS
            ADDRESS)
         (SETQ ENDADR (MKN 'PRINTATOMS))
         LOOP
         (⁚ PRINT THE ATOM, CORRESPONDING TO ADR)
         (PRINT (UNBOX ADR))
         (COND ((EQP ADR ENDADR) (RETURN)))
         (⁚ GET ADDRESS (AS NUMBER) TO NEXT ATOM)
         (SETQ ADR (IPLUS ADR 16))
         (GO LOOP)))
```

30.4 Sorting. There exists a sort function sort, which sort a list of items. The order can be specified by giving a function to sort. A default function alphorder exist. A function merge can merge two sorted lists.

sort[data,comparefn]	Sorts the elements on data. If comparefn is NIL alphorder is used.
merge[a,b,comparefn]	Merges a the sorted lists a and b. Comparefn as sort.
alphorder[a,b]	Returns T if a comes before b, otherwise NIL.

30.5 Control characters[1]

At any time the user may (temporarily) interrupt the computation by pressing the attention key whereupon the system will type ?= to denote that it is ready to receive a so-called attention-command. If the command then entered by the user is for instance the letter H followed by a carriage return we call this sequence of interactions an "attention-H".

The following attention commands exist:

H (interrupt ("Hold it") interrupt at next function call. Interlisp goes into a break.

B (break) computation is stopped, stack backed to last function call, and a break occurs.

E (error) an immediate errorb is generated.

D (reset), ("Dam'n it")) control returns immediately to top level.

O (output) clear output buffer and continue.

T (time) prints CPU time in ms spent and continues.

Pnumber (Print) Sets printlevel to number. If number is followed by any non-numeric character (ie P10.) printlevel will be changed permanently, otherwise only the printlevel for the current printout will be changed.

carriage return without any previous command acts as a no-op. Any other attention command will cause INTERLISP to type ???? and continue.

[1] This part is reproduced from the INTERLISP/360-370 Reference Manual.

Solutions

Section 1

a. literal atom

b. literal atom - in other LISP systems an atom must start with a letter, but in INTERLISP it can begin with any character.

c. floating point number.

d. not correct, to read the atom (you must precede it with %.

e.. list, it is identical to ((NIL))

f. not correct.

g. literal atom, internally the atom ().

h. list.

i. integer.

j. literal atom, the similarly-written floating-point number is 12.E+34.

k. list.

l. string.

m. not correct.

n. not correct, see o.

o. string, consisting of the characters ", blank and ".

p. list, in other LISP systems a , (comma) was treated as an element-separator in a list and was then ignored, but in INTERLISP the , is treated as an ordinary literal atom.

q. literal atom.

r. list, with two strings as elements.

s. list.

t. list, will be printed as (A B (C D) (E))

u. not correct, the brackets match each other, so the right parenthesis does not match anything.

v. not correct, the right parenthesis matches the left bracket, so the right bracket does not match anything.

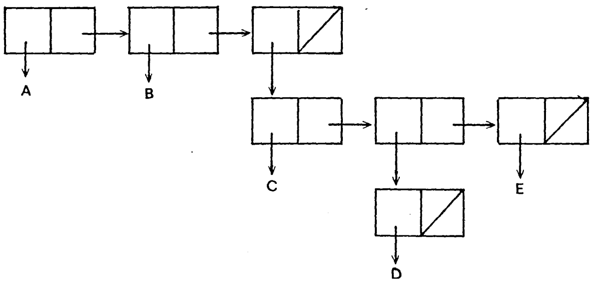
x. not correct, one right parenthesis is missing.

y. list, the end of the list is a dotted-pair, more about this follows in the next section.

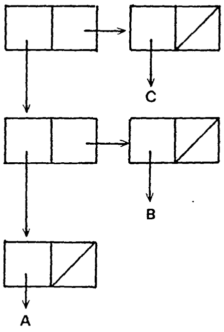
z. list, it has one element A.B

Section 2

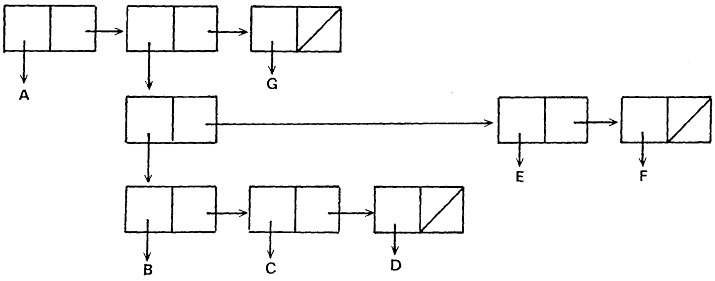
1a. (A B (C (D) E))



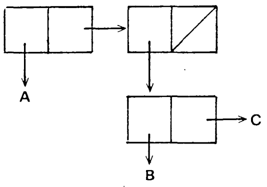
1b. (((A) B) C)



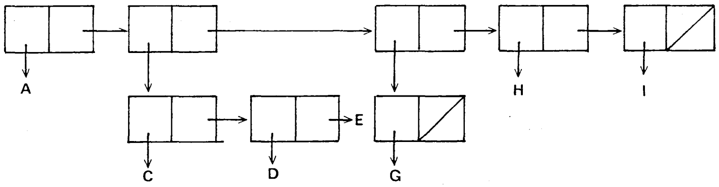
1c. (A <(B C D) E F> F>



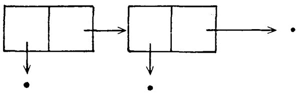
1d. (A (B . C))



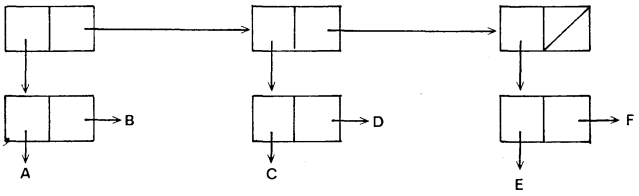
1e. (A (C D . E) (G . NIL) . (H I>



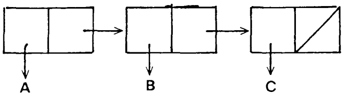
1f. (. . . .)



1g. ((A . B) (C .) (E . F))

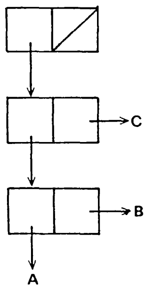


1h. (A . (B . (C . NIL)))



(Solutions, Section 2)

1i. (((A . B) . C) . NIL)



2. The following expressions will be printed in a different way than they were read:

   1c. (A ((B C D) E F) G)

   1e. (A (C D . E) (G) H I)

   1h. (A B C)

   1i. (((A . B) . C))

Section 3

1. atom[caddr[(A B (X Y) C)]] the third element is the list (X Y) and the value is NIL.

2. equal[cadar[((A (Q Q) (A A))], (Q Q)] the value is T.

3. cons[ADAM, cons[(BERTIL), cons[((CAESAR)), NIL]]] the list constructed is (ADAM (BERTIL) ((CAESAR)))

4. a. D
   b. NIL
   c. (B Q Q)
   d. ((= =). =), it will be a dotted-pair at the end of the list.
   e. NIL, for car[NIL] is always NIL, also cdr[NIL] is NIL.
   f. T, see e.
   g. B
   h. (C . D).
   i. (? - ⁑)

5. a. yes, it is <u>true</u>
   b. yes, it is <u>true</u>
   c. no, it is not <u>true</u>
      cons[cdr[l], cdadr[l]] is (((B C) D)) D)

Section 5

1. We give every node n two properties, PRED and SUCC. Corresponding value is a list of nodes, which are predecessors resp successors to the node n.

   The graph can then be stored with

   ```
   put[A,SUCC,(B C)]
   put[B,SUCC,(B D)]
   put[B,PRED,(A B)]
   put[C,SUCC,(D)]
   put[C,PRED,(A D)]
   put[D,SUCC,(C)]
   put[D,PRED,(B C)]
   ```

   The questions can be answered by

   ```
   a. getp[B,SUCC]
   b. getp[C,PRED]
   c. memb[C,getp[A,SUCC]]
   d. if memb[C,getp[A,SUCC]] then
         if memb[C,getp[D,PRED]] then T else NIL
            else NIL
   ```

   or alternatively

   ```
   if memb[C,getp[A,SUCC]] then memb[C,getp[D,PRED]]
   ```

   The first function will return T or NIL, but the other alternative will return a true value or NIL. Section 3.6 describes the value returned by memb.

   ```
   e. if memb[D,getp[C,SUCC]] then memb[C,getp[D,SUCC]]
   f. if memb[B,getp[B,SUCC]] then memb[B,getp[B,PRED]]
   ```

Section 6

```
1a. (CDR '(A B C))
 b. (EQUAL 'A (CAR '((A))))
 c. (ATOM 12.34E4)
 d. (EQUAL L (CONS (CAR L) (CDR L)))
 e. (MEMB 'C (CAADR L))
 f. (PUT (GETP 'KARL 'MARRIED)
         'CHILDREN
         (CONS 'EVA (GETP 'KARL 'CHILDREN>
 g. (COND ((NULL L) NIL)
          (T (CDR L)))
 h. (COND ((EQ (GETP 'ANNE 'MARRIED) 'KARL) T)
          (T (EQ (GETP 'KARL 'MARRIED) 'ANNE>
 i. (COND ((GETP 'JOHN 'FATHER-FATHER))
          ((GETP (GETP 'JOHN 'FATHER) 'FATHER)))
```

Remember that the '-sign is identical to the quote function, so 'A is identical to (QUOTE A).

Section 7

1. (DE CD5R (L) (CADDR (CDDR L)))

   If l has less than 5 elements cd5r will return NIL as value, cdr[NIL] = car[NIL] = NIL

2a.
```
(DE MARRIEDQ (X Y)
    (COND ((EQ X (GETP Y 'MARRIED)) 'YES)
          ((EQ Y (GETP X 'MARRIED)) 'YES)
          (T 'NO)))
```

b.
```
(DE SON (X Y) (PUT X 'FATHER Y)
              (ADDPROP Y 'SON X)
              'OK)
```

   A new property son is introduced.

c.
```
(DE FATHEROF (X Y)
    (COND ((EQ X (GETP Y 'FATHER)) 'YES)
          ((MEMB Y (GETP X 'SON)) 'YES)
          (T 'NO)))
```

3. When the assignments are done the following values exist,

R has value A	R obtained a new value in the assignment (SET L (CAR R))
L has value R	
X has value R	
A has value (Q R S)	A got its value in (SET R '(Q R S))
Q has value (R S)	Q got its value in the last assignment.

   The following expressions are true.

   a,c,d,f,g,i,j

   Remember that an expression is true if its value ≠ NIL.

Section 9

1.
```
(DE EVEN (L) (COND ((NULL L) T)
                   ((NULL (CDR L)) NIL)
                   (T (EVEN (CDDR L)))))
```

2.
```
(DE APPEND2 (X Y) (COND ((NULL X) Y)
                        (T (APPEND2 (CDR X) (CONS (CAR X) Y)))))
```

There is a system function append, which does the same as append2, but which actually is more general. It takes an arbitrary number of lists and concatenates them.

eg append[(A B C), (Q W E), (X)] = (A B C Q W E X)

3.
```
(DE ⁙INTERSECTION (X Y)
  (COND ((NULL X) NIL)
        ((MEMB (CAR X) Y) (CONS (CAR X) (⁙INTERSECTION (CDR X) Y)))
        (T (⁙INTERSECTION (CDR X) Y))))
```

4.
```
(DE ⁙REVERSE (L)
  (COND ((NULL L) NIL)
        (T (APPEND2 (⁙REVERSE (CDR L)) (CONS (CAR L) NIL)))))
```

An alternative solution is

```
(DE ⁙REVERSE (L R)
  (COND ((NULL L) R)
        (T (⁙REVERSE (CDR L) (CONS (CAR L) R)))))
```

We have here introduced an extra argument to ⁙reverse. The reversed list will be built on that list. R must be initialized to NIL. To use ⁙reverse we write

```
(⁙REVERSE '(A B C) NIL)
```

but this is identical to

```
(⁙REVERSE '(A B C))
```

Arguments not given at call will be initialized to NIL automatically. See further in Section 14.

5.
```
(DE ⁙SUBST (X Y L)
  (COND ((NULL L) NIL)
        ((EQUAL Y (CAR L)) (CONS X (⁙SUBST X Y (CDR L))))
        (T (CONS (CAR L) (⁙SUBST X Y (CDR L))))))
```

Suppose l can contain dotted-pairs. The solution is not sufficient then. The following solution will take care of this.

```
(DE ⁑SUBST (X Y L)
  (COND ((EQ L Y) X)
        ((NLISTP L) L)
        ((EQUAL Y (CAR L)) (CONS X (⁑SUBST X Y (CDR L))))
        (T (CONS (CAR L) (⁑SUBST X Y (CDR L))))))
```

⁑subst[NEW, OLD, (A OLD B . OLD) = (A NEW B . NEW)]

6.
```
(DE TOTREVERSE (L)
  (COND ((NLISTP L) L)
        (T (APPEND2 (TOTREVERSE (CDR L)) (CONS (TOTREVERSE (CAR L)
                                                      NIL))))))
```

7.
```
(DE TOTSUBST (X Y L)
  (COND ((NLISTP L) NIL)
        ((EQUAL (CAR L) Y)
                (CONS X (TOTSUBST X Y (CDR L))))
        (T (CONS (TOTSUBST X Y (CAR L))
                 (TOTSUBST X Y (CDR L>
```

8.
```
(DE ⁑SUBLIS (AL L)
   (COND ((NULL AL) L)
         (T (⁑SUBLIS (CDR AL) (TOTSUBST (CDAR AL) (CAAR AL) L)))))
```

An alternative solution is

```
(DE ⁑SUBLIS (AL L)
    (COND ((NLISTP L) (COND ((SETQ TEMP (⁑ASSOC L AL)) (CDR TEMP))
                            (T L)))
          (T (CONS (⁑SUBLIS AL (CAR L))
                   (⁑SUBLIS AL (CDR L))))))
```

with ⁑assoc defined as

```
(DE ⁑ASSOC (X AL) (COND ((NULL AL) NIL)
                        ((EQ X (CAAR AL)) (CAR AL))
                        (T (⁑ASSOC X (CDR AL)))))
```

assoc searches an association list for the first pair, whose car is equal to x, and returns that pair.

eg *assoc[Q, ((A . B) (Q . QQ) (Q . W))] = (Q . QQ)

TEMP is introduced for temporary hold of the value returned from *assoc. If the value was true we return cdr of that pair.

The two solutions have different strategies. The first takes one pair each time and scans through the list and substitutes. The list will be scanned as many times as there are pairs.

The second solution takes one element at a time and scans through the pairs for checking if that element shall be substituted and if it substitutes. The list of pairs will be scanned as many times as there are elements on l. So we can choose strategy depending on the length of the lists.

9.
```
(DE PAIR (X Y)
    (COND ((NULL X) NIL)
          (T (CONS (CONS (CAR X) (CAR Y))
                   (PAIR (CDR X) (CDR Y))))))
```

We test only if x is empty, y must then be empty depending on the assumption.

10.
```
(DE FLATTEN (L)
   (COND ((NLISTP L) L)
         ((ATOM (CAR L)) (CONS (CAR L) (FLATTEN (CDR L))))
         (T (APPEND2 (FLATTEN (CAR L)) (FLATTEN (CDR L))))))
```

append2 is defined in Exercise 2.

An alternative is

```
(DE FLATTEN (X Y)
   (COND ((NULL X) Y)
         ((NLISTP X) (CONS X Y))
         (T (FLATTEN (CAR X) (FLATTEN (CDR X) Y)))))
```

This definition is harder to understand. Run it on a computer and trace it. The "flattened" list will be built up in y backwards. Of the above function INTERLISP already contains intersection, reverse, subst, sublis and assoc.

11. Start with order.

```
(DE ORDER (X Y) (MEMB Y (MEMB X PRECEDENCE)))
```

The binary tree can be constructed of nodes where every node is a list of three elements

(data-element pointer-to-left-subtree pointer-to-right-subtree)

The tree in the example should then be represented as

```
(D (B (A NIL NIL) (C NIL NIL))
   (E NIL (J NIL NIL)))
```

For convenience and readability we introduce some small help functions.

```
(DE DATA (NODE) (CAR NODE))
(DE LEFT (NODE) (CADR NODE))
(DE RIGHT (NODE) (CADDR NODE))
(DE MAKENODE (DATA LEFTTREE RIGHTTREE)
  (CONS DATA (CONS LEFTTREE (CONS RIGHTTREE NIL))))
```

The next function to define is <u>buildtree</u>

```
(DE BUILDTREE ( L TREE)
  (COND ((NULL L) TREE)
        (T (BUILDTREE (CDR TREE)
                      (INSERTNODE (MAKENODE (CAR L) NIL NIL)
                                  TREE)))))
```

<u>tree</u> is here the argument, which initialized to NIL and which is used for the generated tree.

<u>Insertnode</u> takes a node and puts it in the tree.

```
(DE INSERTNODE (NODE TREE)
  (COND ((NULL TREE) NODE)
        ((ORDER (DATA NODE) (DATA TREE))
                   (MAKENODE (DATA TREE)
                             (INSERTNODE NODE (LEFT TREE))
                             (RIGHT TREE)))
        (T (MAKENODE (DATA TREE)
                     (LEFT TREE)
                     (INSERTNODE NODE (RIGHT TREE))
           ))))
```

With the new node we will follow its path through the tree, actually we make new nodes for the nodes we are passing. When we have come to a terminal node the new node will be inserted there.

These functions will now create the tree and walktree willl traverse it and make a list of the data elements in the nodes. We will traverse the tree in postorder traversal. This means that we are going to the left subtree first, then collect the node and at last to the right subtree.

```
(DE WALKTREE (TREE)
  (COND ((NULL TREE) NIL)
        (T (APPEND2 (WALKTREE (LEFT TREE))
                    (CONS (DATA TREE) (WALKTREE (RIGHT TREE)))
           ))))

(DE TREESORT (L) (WALKTREE (BUILDTREE L)))
```

These functions clearly illustrate the recursive approach of describing algorithms. In a non-recursive language these functions would be much longer and more difficult to read. This way of splitting the functions is very common in LISP.

Instead of introducing the small help functions, by defining them as function call to its equivalent system function, we can use a function movd, which moves a function definition to another function name. The functions will be absolutely identical and it costs no more to have done it. We have inexpensively introduced a synonym.

```
movd[CAR,GETDATA]
movd[CADR,GETLEFT]
movd[CADDR,GETRIGHT]
movd[LIST,MAKENODE]      this is a function list, for making
                         conses like these. See next section.
```

Observe that it is only these functions which really know the structure of a node. So, if we want to change a node to

ie (data-element left subtree . right subtree)

which is more compact, takes only two list cells against three list cells with the other representation. We need only to redefine getright and makenode.

The functions described here are rather list cell consuming. There are other functions, which we could have used for saving list cells. See further in Section 21.

The function merge can be defined as

```
        (DE MERGE (X Y)
          (COND ((NULL X) Y)
                ((NULL Y) X)
                ((ORDER (CAR X) (CAR Y)) (CONS (CAR X) (MERGE (CDR X) Y)))
                (T (CONS (CAR Y) (MERGE X (CDR Y))))))
```

Section 11

```
1.  (DE POINTLIST (HAND)
      (SELECTQ (CDAR HAND)
               (NIL NIL)
               (ACE (CONS (SELECTQ (CAAR HAND)
                                   (SPADE 10)
                                   (HEART 9)
                                   (DIAMOND 8)
                                    7)
                          (POINTLIST (CDR HAND))))
               (KING (CONS 5 (POINTLIST (CDR HAND))))
               ((QUEEN JACK) (CONS 3 (POINTLIST (CDR HAND))))
               ((7 3) (CONS 1 (POINTLIST (CDR HAND))))
               (POINTLIST (CDR HAND))))

2.  (DE PROG2 (X Y) Y)

3.  (DE ::MEMBER (X L) (COND ((NULL L) NIL)
                             ((EQUAL X (CAR L)) L)
                             (T (::MEMBER X (CDR L)))))

    (DE ::LAST (L) (COND ((NULL L) NIL)
                         ((NULL (CDR L)) L)
                         (T (::LAST (CDR L)))))

4.  (DE ::ADDPROP (ATM PROP NEW FLG TEMPVAL)
      (SETQ TEMPVAL (GETP ATM PROP))
      (PUT ATM PROP (SELECTQ FLG
                             (T (CONS NEW TEMPVAL))
                             (REVERSE (CONS NEW (REVERSE TEMPVAL))))))
```

There are more efficient ways of adding an element to the end of a list than making reverse twice. The function nconc1 makes this, see further Section 21.

The variable tempval is introduced for holding a temporary value. Its appearance in the variable list makes the variable to a local variable to this function. This is more comprehensibly discussed in Section 15.

```
5.  (DE ::DEFLIST (ATM-VAL PROP)
      (COND ((NULL ATM-VAL) NIL)
            (T (PUT (CAAR ATM-VAL) PROP (CADAR ATM-VAL))
               (CONS (CAAR ATM-VAL) (DEFLIST (CDR ATM-VAL) PROP)))))

    a.  (DE ::GET (FREEPROP PROP) (CADR (MEMB PROP FREEPROP)))

    b.  (DE PUTF (FREEPROP PROP VAL)
          (COND ((NULL FREEPROP) (LIST PROP VAL))
                ((EQ (CAR FREEPROP) PROP) (CONS PROP (CONS VAL FREEPROP)))
                (T (CONS (CAR FREEPROP)
                         (CONS (CADR FREEPROP)
                               (PUTF (CDDR FREEPROP) PROP VAL))))))

6.  (DE ::SQCDR (L) (PROG1 (CAR L) (SETQ L (CDR L))))

7.  a.  (DE ::ASSOC (A AL)
          (COND ((NULL AL) NIL)
                ((EQ A (CAAR AL)) (CAR AL))
                (T (::ASSOC A (CDR AL)))))

    b.  (DE CHASSOC (AL A NEW)
          (COND ((NULL AL) NIL)
                ((EQ A (CAAR AL)) (CONS (CONS A NEW) (CDR AL)))
                (T (CONS (CAR AL) (CHASSOC (CDR AL) A NEW)))))

    c.  (DE REPASSOC (AL A)
          (COND ((NULL AL) NIL)
                ((EQ A (CAAR AL)) (REPASSOC (CDR AL) A))
                (T (CONS (CAR AL) (REPASSOC (CDR AL) A)))))
```

Section 12

```
1.  (DE *LENGTH (L) (COND ((NULL L) 0)
                          (T (ADD1 (*LENGTH (CDR L))))))

2.  (DE FAK (N) (COND ((IZEROP N) 1)
                      (T (ITIMES N (FAK (SUB1 N))))))

3.  (DE POINTS (L) (COND ((NULL L) 0)
                         (T (IPLUS (CAR L) (POINTS (CDR L))))))

4.  (DE DIFF (EXPR X)
      (COND ((EQ EXPR X) 1)
            ((ATOM EXPR) 0)
            (T (SELECTQ (CAR EXPR)
                        (PLUS (DERPLUS EXPR X))
                        (DIFFERENCE (DERDIFF EXPR X))
                        (TIMES (DERTIMES EXPR X))
                        (QUOTIENT (DERQUOTIENT EXPR X))
                        (MINUS (DERMINUS EXPR X))
                        (EXPT (DEREXPT EXPR X))
                        (SIN (DERSIN EXPR X))
                        (COS (DERCOS EXPR X))
                        EXPR))))

    (DE DERPLUS (EXPR X) (CONS 'PLUS (MAPPLUS (CDR EXPR) X)))

    (DE MAPPLUS (L X) (COND ((NULL L) NIL)
                            (T (CONS (DIFF (CAR L) X) (MAPPLUS (CDR L)
                                                                 X)))
                            )))

    (DE DERDIFF (EXPR X) (LIST 'DIFFERENCE
                               (DIFF (CADR EXPR) X)
                               (DIFF (CADDR EXPR) X)))

    (DE DERTIMES (EXPR X)
      (LIST 'PLUS
            (LIST 'TIMES (CADR EXPR) (DIFF (CADDR EXPR) X)
            (LIST 'TIMES (CADDR EXPR) (DIFF (CADR EXPR) X)))
```

```
(DE DERQUOTIENT (EXPR X)
  (LIST 'QUOTIENT
        (LIST 'DIFFERENCE
              (LIST 'TIMES (CADDR EXPR) (DIFF (CADR EXPR) X))
              (LIST 'TIMES (CADR EXPR) (DIFF (CADDR EXPR) X)))
        (LIST 'EXPT (CADR EXPR) 2)))

(DE DERMINUS (EXPR X) (LIST 'MINUS (DIFF (CADR EXPR X))))

(DE DEREXPT (EXPR X)
  (LIST 'TIMES
        (CADDR EXPR)
        (LIST 'TIMES
              (DIFF (CADR EXPR) X)
              (LIST 'EXPT (CADR EXPR) (SUB1 (CADDR EXPR))))))

(DE DERSIN (EXPR X)
  (LIST 'TIMES
        (LIST 'COS (CADR EXPR))
        (DIFF (CADR EXPR X)))

(DE DERCOS (EXPR X)
  (LIST 'MINUS
        (LIST 'TIMES
              (LIST '$IN (CADR EXPR))
              (DIFF (CADR EXPR) X))))
```

4. There is no answer to this exercise! The best way to write simplification function is to make use of the computer, test them and find the different simplification rules. There is however, a solution described in Weissman's LISP 1.5 Primer (ref 5).

Section 13

1.
```
(DE EVEN (L) (OR (NOT L) (AND (CDR L) (EVEN (CDDR L)))))
```

2.
```
(DE MAZE (L IN OUT WAY)
  (MAZ L (CDR (ASSOC IN L)) OUT (CONS IN WAY)))

(DE MAZ (L INS OUT WAY)
  (COND ((NULL INS) NIL)
        ((EQ (CAR INS) OUT) (REVERSE (CONS OUT WAY)))
        ((MEMB (CAR INS) WAY) (MAZ L (CDR INS) OUT WAY))
        (T (OR (MAZE L (CAR INS) OUT WAY)
               (MAZ L (CDR INS) OUT WAY)))))
```

way is an extra argument used for building the way we are trying to find.

Section 14

Warning - If you intend to test these functions on the computer, use the ⁚⁚-name of the function. A system function defined incorrectly can clobber the system!

1.
```
(PUTD 'DF '(NLAMBDA (FN . L) (PUTD FN (CONS 'NLAMBDA L>
```

2.
```
(DE ::APPEND L (COND ((NULL L) NIL)
                     ((NULL (CDR L)) (APPEND2 (CAR L) NIL))
                     (T (APPENDN L))))
(DE APPENDN (L) (COND ((NULL (CDR L)) (CAR L))
                       (T APPEND2 (CAR L) (APPENDN (CDR L))))))
(DE APPEND2 (X Y) (COND ((NULL X) Y)
                         (T (CONS (CAR X) (APPEND2 (CDR X) Y))))))
```

The system function ::append copies the top level of the n-1 first lists and concatenates this to the last list. But if ::append only has one list as argument that list is copied on top level.

3.
```
(DF / L L)
```

4.
```
(DF ::AND L (COND ((NULL L) T)
                  (T (AND1 L))))
(DE AND1 (L) (COND  ((NULL (CDR L)) (EVAL (CAR L)))
                    ((EVAL (CAR L)) (AND1 (CDR L)))
                    (T NIL)))
```

5.
```
(DF ::SELECTQ (A . L) (::SELECTQ1 A L))

(DE ::SELECTQ1 (A L) (COND ((NULL CDR L)) (EVAL (CAR L)))
                           ((OR (AND (ATOM (CAAR L)) (EQ A
                                                  (CAAR L)))
                                 (AND (LISTP (CAAR L)) (MEMB A
                                                  (CAAR L))))
                                 (EVPROGN (CDAR L)))
                           (T (::SELECTQ1 A (CDR L)))))

(DE EVPROGN (L) (COND ((CDR L) (EVAL (CAR L)) (EVPROGN (CDR L)))
                            (T (EVAL (CAR L)))))
```

Evprogn takes a list of forms and evaluates them, returns the value of the last form.

6.
```
(DF IF L (COND ((EVAL (CAR L)) (EVAL-TO-ELSE (CDDR L)))
               (T (EVPROGN (CDR (MEMB 'ELSE (CDDR L)))))))
(DE EVAL-TO-ELSE (L)
  (COND ((OR (NULL (CDR L)) (EQ (CADR L) 'ELSE)) (EVAL (CAR L)))
        (T (EVAL (CAR L)) (EVAL-TO-ELSE (CDR L)))))
```

The function works also without a _then_- or an _else_ branch.

```
(DF DO L (DOA (FORMS-TO-UNTIL L) (CAR (LAST L))))

(DE FORMS-TO-UNTIL (L)
  (COND ((EQ (CAR L) 'UNTIL) NIL)
        (T (CONS (CAR L) (FORMS-TO-UNTIL (CDR L))))))

(DE DOA (FORMS TEST)
  (EVLIS FORMS) (AND (NOT (EVAL TEST)) (DOA FORMS TEST)))

(DE EVLIS (L) (COND ((NULL L) NIL)
                    (T (EVAL (CAR L)) (EVLIS (CDR L)))))
```

evprogn is defined in Exercise 5.

Section 16

```
1.  (DE EVEN (L) (PROG NIL
        LOP
        (COND ((NULL L) (RETURN T))
              ((NULL (CDR L)) (RETURN NIL)))
        (SETQ L (CDDR L))
        (GO LOP)))

2.  (DE APPEND2 (X Y) (PROG NIL
         (SETQ X (REVERSE X))
         LOP
         (COND ((NULL X) (RETURN Y)))
         (SETQ Y (CONS (CAR X) Y)))
         (SETQ X (CDR X))
         (GO LOP)))

3.  (DE ::INTERSECTION (X Y) (PROG (VAL)
        LOP
        (COND ((NULL X) (RETURN VAL)
              ((MEMB (CAR X)) Y) (SETQ VAL (CONS (CAR X) VAL))))
        (SETQ X (CDR X))
        (GO LOP)))

4.  (DE ::REVERSE (L) (PROG (VAL)
        TOP
        (COND ((NULL L) (RETURN VAL)))
        (SETQ VAL (CONS (CAR L) VAL))
        (SETQ L (CDR L))
        (GO TOP)))

5.  (DE ::SUBST (X Y L) (PROG (P)
        LOP
        (COND ((NULL L) (RETURN (REVERSE P)))
              ((EQ (CAR L) Y) (SETQ P (CONS X P)))
              (T (SETQ P (CONS (CAR L) P))))
        (SETQ L (CDR L))
        (GO LOP)))
```

6.
```
(DE TOTREVERSE (L) (PROG (X)
  TOP
  (COND ((NULL L) (RETURN X))
        ((LISTP (CAR L)) (SETQ X (CONS (TOTREVERSE (CAR L)) X)))
        (T (SETQ X (CONS (CAR L) X))))
  (SETQ L (CDR L))
  (GO TOP)))
```

Notice that we must make a recursive call in one direction - in this case the car-direction. We can of course write without any recursive call but we must then organize some kind of stack.

Section 17

1. `(DE CALC (OP A B) (APPLY:: OP A B))`

2. `(APPLY 'SET '(A B))`

   The arguments A and B, evaluated will be given to set, which will give A the value B.

   `(APPLY 'SETQ '(E F))`

   The arguments E and F will be given to setq, but setq will itself evaluate by eval the atom F. F has no value and the message U.B.A F will be printed.

   `(APPLY 'SETQQ '(I J))`

   The arguments I and J will be given to setqq, which will give I the value J.

   Set and setqq behave exactly in the same manner when given to apply. Although set is an eval-function and setqq a noeval-function no arguments in the argument list will be evaluated.

3.
   ```
   (DE FIRST (L FN)
     (COND ((NULL L) NIL)
           ((APPLY:: FN (CAR L)) (CAR L))
           (T (FIRST (CDR L) FN))))
   ```

Section 18

1.
```
(DE ::MAP (MAPX MAPFN1 MAPFN2) (PROG NIL
     (COND ((NULL MAPFN2) (SETQ MAPFN2 'CDR)))
     LOP
     (COND ((NULL MAPX) (RETURN NIL)))
     (APPLY:: MAPFN1 MAPX)
     (SETQ MAPX (APPLY:: MAPFN2 MAPX))
     (GO LOP)))

(DE ::MAPCAR (MAPX MAPFN1 MAPFN2)
   (COND ((NULL MAPX) NIL)
         (T CONS (APPLY:: MAPFN1 (CAR MAPX))
                 (::MAPCAR (COND (MAPFN2 (APPLY:: MAPFN2 MAPX))
                                 (T (CDR MAPX)))
                           MAPFN1
                           MAPFN2)))))

(DE ::MAP2C (MAPX MAPY MAPFN1 MAPFN2) (PROG NIL
             (OR MAPFN2 (SETQ MAPFN2 'CDR))
             LOP
             (OR (AND MAPX MAPY) (RETURN NIL))
             (APPLY:: MAPFN1 (CAR MAPX) (CAR MAPY))
             (SETQ MAPX (APPLY:: MAPFN2 MAPX))
             (SETQ MAPY (APPLY:: MAPFN2 MAPY))
             (GO LOP)))

(DE ::EVERY (MAPX MAPFN1 MAPFN2)
     (COND ((NULL MAPX) T)
           ((APPLY:: MAPFN1 (CAR MAPX))
                (::EVERY (COND (MAPFN2 (APPLY:: MAPFN2 MAPX))
                               (T (CDR MAPX))) MAPFN1 MAPFN2))))
```

There are of course many ways of writing <u>map</u>-functions and these are some representative solutions.

2.
```
(DE SQUARE (L) (MAPCAR L (FUNCTION (LAMBDA (X)
                          (TIMES X X)))))
```

```
3.  (DE PAIR (X Y) (MAPC2CAR X Y (FUNCTION CONS))

    (DE COLLECTPAIR (AL A)
      (PROG (RES)
            <MAPC AL (FUNCTION (LAMBDA (X)
                                     (COND ((EQ (CAR X) A)
                                           (SETQ RES (CONS X RES>
            (RETURN RES)))
```

Section 19

```
1.  (DE PASCAL (N)
      (PROG ((INDENT 26) X OLD)
             (:: TEST IF N BETWEEN 0 AND 10)
             (COND ((OR (GREATERP N 10) (LESSP N 0))
                         (PRIN1 '"N NOT IN INTERVAL")
                         (PRIN1 '"N = ") (PRINT N) (RETURN)))
             (SPACES 18)
             (:: PRINT HEADING)
             (PRIN1 '"PASCAL'S TRIANGLE")
             (TERPRI)
             (SPACES 18) (PRIN1 '"-----------------"
             (TERPRI) (TERPRI)'
             (:: PRINT TOP OF TRIANGLE)
             (SPACES INDENT) (PRINT 1)
             (SETQ OLD (LIST 1))
             (:: OLD IS USED TO SAVE THE NUMBERS ON THE LAST LINE PRINTED)
         TOP (COND ((ZEROP N) (TERPRI) (TERPRI) (RETURN)))
             (SETQ INDENT (IDIFFERENCE INDENT 2))
             (:: X CONTAINS THE NUMBERS OF THE LAST LINE AND IS USED FOR
                 CALCULATING THE NEW NUMBERS)
             (SETQ X (CONS 0 OLD))
             (SETQ OLD NIL)
             (SPACES INDENT)
             (:: PREPARATION DONE FOR NEXT LINE)
         LOP (COND ((NULL (CDR X)) (:: PRINT THE LAST 1 AND THE LINE
                                                IS FINISHED)
                                    (PRINT 1)
                                    (SETQ OLD (CONS 1 OLD))
                                    (SETQ N (SUB1 N))
                                    (GO TOP)))
             (:: CALCULATE THE NEW NUMBER FROM THE TWO FIRST NUMBER ON X,
                   PRINT IT AND SAVE IT ON OLD)
             (SETQ OLD (CONS (PRIN1 (IPLUS (CAR X) (CADR X))) OLD))
             (SETQ X (CDR X))
             (SPACES 3)
```

If this statement is replaced by

```
  (SPACES (IDIFFERENCE 4 (NCHARS (CAR OLD))))
```

the triangle will be printed symmetrical. The function nchars[x] gives the number of characters that will be printed if x is printed by prin1. The number of spaces depends on the number of characters in the number just printed. See further Section 22.

```
             (GO LOP)))
```

2. We assume an algol version, where the following characters are used as separators:

   + - ⁑ / : ; , = ( )

   Their internal codes are resp

   78, 96, 92, 97, 122, 94, 107, 126, 77 and 93

   Some combinations of separators should be considered as one entity, such as := .

   It is easy to extend these routines to also take care of other symbols, such that ∨ , ∧ and ¬. Even here there are combinations eg ¬ =.

```
(DE ALGOLSCAN NIL
  (PROG ((OLDBREAK (GETBRK)) L SYM TEMP)
        (SETBRK '(78 96 92 97 122 94 107 126 77 93))
        LOP
        (SETQ SYM (RATOM))
        (SELECTQ SYM
                 (: (⁑ CHECK FOR :=)
                    (SETQ TEMP SYM)
                    (SETQ SYM (RATOM))
                    (SELECTQ SYM
                             (= (SETQ SYM ':=))
                             (SETQ L (CONS TEMP L)))
                 NIL))
        (COND ((EQ SYM 'ENDALGOL) (SETBRK OLDBREAK)
                                  (RETURN (REVERSE L))))
        (SETQ L (CONS SYM L))
        (GO LOP)))

 (ALGOLSCAN)
 BEGIN INTEGER X;REAL Y,Z;
 Z:=10.5;
 TOP: X:=READ;
 IF X=10 THEN GOTO OUT;
 Y := Z⁑(X-12)/Y+12.3;
 GOTO TOP;
 OUT: PRINT(Y);
 END;
 ENDALGOL
```

The following list will be returned

```
(BEGIN INTEGER X ; REAL Y , Z ; Z := 10.5 ; TOP : X := READ ;
IF X = 10 THEN GOTO OUT ; Y := Z * %( X - 12 %) / Y + 12.3
; GOTO TOP ; OUT : PRINT %( Y %) ; END ;)
```

To read in we could have done

```
(SETQ L (RATOMS 'ENDALGOL))
```

which gives a list of all symbols. We could then go through _l_ and look for pairs of symbols which should be combined.

Section 21

1.
```
(DE ::DREMOVE (X L)
  (COND ((NLISTP L) L)
        ((EQ (CAR L) X) (::DREMOVE X (CDR L)))
        (T (RPLACD (RPLACA L (::DREMOVE X (CAR L)))
                   (::DREMOVE X (CDR L))))))
```

Observe how the recursion in car and cdr direction uses rplaca and rplacd.

2.
```
(DE ::DSUBST (X Y L)
  (COND ((EQ Y L) X)
        ((NLISTP L) L)
        (T (RPLACA L (RPLACA L (::DSUBST X Y (CAR L)))
                     (::DSUBST X Y (CDR L))))))
```

3.
```
(DE ::DREVERSE (L)
  (COND ((NULL L) NIL)
        (T (NCONC (::DREVERSE (CDR L)) (RPLACD L NIL)))))
```

4.
```
(DE ::ADDPROP (ATM PROP NEW FLG)
  (COND ((NULL (CDR ATM)) (CADDR (RPLACD ATM (LIST PROP (LIST
                                                         NEW)))))
        ((EQ PROP (CADR ATM)) (CAR (RPLACA (CDR ATM)
                     (SELECTQ FLG
                              (T (CONS NEW (CADR ATM)))
                              (NCONC1 (CADDR ATM) NEW)))))
        (T (::ADDPROP (CDDR ATM) PROP NEW FLG)))))
```

In the first test the caddr is made for obtaining the correct value.

5.
```
(DE ::LCONC (PTR L)
   (COND ((NULL PTR) (SETQ PTR (CONS NIL NIL))))
   (COND ((NULL (CAR PTR) (RPLACA PTR L)
                          (RPLACD PTR (LAST L)))
         (T (RPLACD PTR
                    (LAST (RPLACD (CDR PTR) L>
```

We start to set up a pointer cell if it does not exist, then if this is the first element set the begin pointer and lastly concatenate the new list at the end and update the end pointer.

6. (DE *ATTACH (X Y)

```
   (RPLACA (RPLACD Y (CONS (CAR Y) (CDR Y))) X)
```

7. We redefine <u>insertnode</u>

```
   (DE INSERTNODE (NODE TREE)
     (COND ((NULL TREE) NODE)
           (T (RPLACA (SEARCHTREE TREE NODE)
                      NODE)
              TREE)))
```

If the tree is empty <u>node</u> is returned as the root of the tree, otherwise <u>searchtree</u> finds the terminal node, to which <u>node</u> is connected. But tree, the pointer to the root of the tree, is returned as value.

<u>Searchtree</u> is defined as

```
   (DE SEARCHTREE (TREE NODE)
     (COND ((ORDER (GETDATA NODE)
                   (GETDATA TREE))
            (COND ((NULL (GETLEFT TREE)) (LEFTP TREE))
                  (T (SEARCHTREE (GETLEFT TREE) NODE))))
           ((NULL (GETRIGHT TREE) (RIGHTP TREE))
           (T (SEARCHTREE (GETRIGHT TREE) NODE))))
```

<u>Leftp</u> and <u>rightp</u> are introduced for readability in the same way as <u>getdata</u>, <u>getleft</u> etc.

```
   (MOVDQQ CDR LEFTP)
   (MOVDQQ CDDR RIGHTP)
```

Section 22

1.
```
(DE FILENAME (FILE) (FILENAME1 (UNPACK FILE)))
(DE FILENAME1 (L RES)
  (COND ((NULL L) FILE)
        ((EQ (CAR L) '≠) (PACK RES))
        (T (FILENAME1 (CDR L) (NCONC1 RES (CAR L))))))
```

Without the ≠ in the filename we can return the original filename. This is done by filename1 through returning the free variable file.

An alternative is to use nthchar and directly check if the third character from the end is ≠ .

```
(DE FILENAME (FILE)
  (COND ((NEQ '≠(NTHCHAR FILE -3)) FILE)
        (T (FILENAME1 (UNPACK FILE)))))
```

Using string functions

```
(DE FILENAME (FILE) STRF)
  (SETQ STRF (MKSTRING FILE))
  (COND ((STREQUAL "≠" (SUBSTRING STRF -3 -3))
                            (GLC STRF)
                            (GLC STRF)
                            (GLC STRF)
                            STRF)
         (T FILE)))
```

2.
```
(DE STRPOS (SUBSTR STR)
  (PROG ((LSTR (NCHARS STR))
        (MIN 1)
        MAX)
        (⁑ USE SUBSTRING ON STR FOR GETTING SUCCESSIVE SUBSTRINGS.
           MIN AND MAX ARE LIMITS FOR THE SUBSTRING)
        (SETQ MAX (NCHARS SUBST))
        LOP
        (COND ((IGREATERP MAX LSTR) (⁑ NOMATCH) (RETURN NIL))
              ((STREQUAL (SUBSTRING STR MIN MAX) SUBSTR)
                                   (⁑ MATCH) (RETURN MIN)))
        (SETQ MIN (ADD1 MIN))
        (SETQ MAX (ADD1 MAX))
        (GO LOP)))
```

Section 23

1. The index function, which will be generated, can be found in Knuth's Fundamental Algorithms. Given a k-dimensional array A with elements

   $A[I_1, I_2, \ldots, I_k]$ for

   $1 \leq I_1 \leq d_1,\ 1 \leq I_2 \leq d_2, \ldots .\ 1 \leq I_k \leq d_k$ the index function is

   $$\mathrm{INDEX}[I_1, I_2, \ldots, I_k] = \sum_{1 \leq r \leq k} a_r I_r \div \sum_{1 \leq r \leq k-1} a_r$$

   where

   $$a_r = \prod_{r < s \leq k} d_s$$

   The example will generate an index function

   INDEX[I,J,K] = 35*I + 7*J + K - 42

   or a LISP expression

   ```
   (LAMBDA (I J K) (IPLUS (ITIMES 35 I) (ITIMES 7 J) K -42))
   ```

   Some auxiliary functions are introduced and explained later.

   ```
   (DF DEFARRAY (ARR . INDICES)
     (PROG (LAMBDAVARS INDEXFN)
           (* ALLOCATE SPACE AND ASSIGN THE ARRAY POINTER)
           (SET ARR (ARRAY (MULT INDICES)))
           (* GET THE CORRECT NUMBER OF LAMBDAVARIABLES)
           (SETQ LAMBDAVARS (VARS INDICES
                                  '(I J K L M N O P)))
           (* CONSTRUCT INDEX FUNCTION)
           (SETQ INDEXFN (GENINDEX INDICES LAMBDAVARS))
           (* STORE INDEX FUNCTION ON THE ARRAYS PROPERTY LIST
              UNDER PROPERTY INDEXFN)
           (PUT ARR 'INDEXFN (LIST 'LAMBDA LAMBDAVARS INDEXFN))
           (* CONSTRUCT AND STORE ACCESS FUNCTION)
           (PUTD ARR (LIST 'LAMBDA
                           LAMBDAVARS
                           (LIST 'ELT ARR INDEXFN)))
           (RETURN ARR>
   ```

   One weakness in this design is that the array names must differ from function and variable names that are in the system.

```
(DE GENINDEX (INDS VARS)
  (PROG ((L (CONS)))
        (:: L IS USED BY TCONC AS A QUEUE POINTER)
        (:: SIMPLE IF ONE-DIMENSIONAL)
        (COND ((NULL (CDR VARS)) (RETURN (CAR VARS))))
        (:: GO LOOPING UNTIL THE INDEXFUNCTION IS BUILT)
        LOP
        (TCONC L (LIST 'ITIMES (MULT (CDR INDS)) (CAR VARS)))
        (SETQ INDS (CDR INDS))
        (SETQ VARS (CDR VARS))
        (COND ((CDR VARS) (GO LOP)))
        (TCONC L (CAR VARS))
        (TCONC L (IMINUS (CONSTANT (CAR L))))
        (RETURN (CONS 'PLUS (CAR L>
```

<u>Mult</u> multiplies the numbers in a list.

```
mult[(3 5 7)] = 105
(DE MULT (L) (COND ((NULL L) 1)
                   (T (ITIMES (CAR L) (MULT (CDR L>
```

<u>Vars</u> returns a list of the <u>k</u>th first variables on a variable list, where <u>k</u> is the number of indices.

```
(DE VARS (INDS VARLIST)
  (COND ((NULL INDS) NIL)
        (T (CONS (CAR VARLIST) (VARS (CDR INDS) (CDR VARLIST>
```

<u>Constant</u> calculates the constant in the index function. It goes through the generated index expression and adds the calculated $\underline{a}_r$.

```
(DE CONSTANT (L)
  (COND ((NULL (CDR L)) 0)
        (T (IPLUS (CADAR L) (CONSTANT (CDR L>
```

<u>Setarray</u> is defined by

```
(DF SETARRAY (ARRPOS VAL)
  (:: A NO-EVAL FUNCTION)
  (SETA (EVAL (CAR ARRPOS))
        (APPLY (GETP (CAR ARRPOS) 'INDEXFN)
               (MAPCAR (CDR ARRPOS) 'EVAL))
        (EVAL VAL>
```

Notice how we must evaluate car[arrpos] in order to get the array pointer, and also that we evaluate the indices and the value.

Section 25

The choice function can be defined as

```
(DF CHOICE (VAR VALLIST FORM)
    (PROG (VALUE)
          LOP
          (COND ((NULL VALLIST) (*ALL VALUES USED, REPORT FAILURE)
                                (FAILURE)))
          (SET VAR (CAR VALLIST))
          (SETQ VALLIST (CDR VALLIST))
          (* EVALUATE THE FORM WITH ERRORSET)
          (SETQ VALUE (ERRORSET FORM))
          (COND ((NULL VALUE) (* BACKTRACKING HAS OCCURRED,
                                 MAKE A NEW CHOICE)
                              (GO LOP))
                (T (SUCCESS (CAR VALUE>
```

The failure and success functions can be defined by

```
(MOVDQQ ERRORB FAILURE)
(MOVDQQ RETURN SUCCESS)
```

The 8-queen problem can now be stated as

```
(DE QUEEN (ROW BOARD)
  (PROG (COL) (RETURN
     (COND ((EQ ROW 9) (* ALL QUEENS ARE NOW PLACED ON BOARD)
                       (PRINTBOARD)
                       (SUCCESS 'DONE))
           (T (CHOICE COL
                      (1 2 3 4 5 6 7 8)
                      (COND ((TESTBOARD) (* OK TO PLACE QUEEN
                                            IN THIS COLUMN)
                                         (QUEEN (ADD1 ROW)
                                                (PLACEBOARD)))
                            (T (* CAN NOT PLACE HERE)
                               (FAILURE>
```

(Solutions, Section 25)

If the problems are without solutions we will "errorbang" out through the function 8-queen and return to the top-loop. A better solution is to catch this "errorbang" simply by putting an errorset in 8-queen. We redefine it to

```
(DE 8-QUEEN NIL (PROG (VAL)
                 (SETQ VAL (ERRORSET '(QUEEN 1 (STARTBOARD)))))
                 (RETURN (COND (VAL (CAR VAL))
                               (T 'FAILURE>
```

These two functions are defined to be independent of the implementation of the board. Startboard initializes a board, printboard prints the board, testboard checks if a queen can be placed safe at the position given by row and col and returns the T, otherwise NIL, and placeboard places the new queen on the board. The functions are without arguments but they can use row, board and col as free variables.

The board could be implemented as an 8x8 array or as a list with 64 elements. A simpler representation is a list of those columns where queens already have been placed. Let us look at this representation a little closer and later discuss a problem concerning the other mentioned representations.

Suppose on the first row a queen is placed in column one, on the second row a queen in column three and on the third row a queen in column five. The board is then a list (5 3 1). To test if a new queen placed on (row, col) is safe from attack from queens already placed, the following tests are made

a. Is the column already occupied? Is col in the board list.

b. Is the diagonal already accupied? Take the first element c in the board list and compare if col is equal to c $\pm$ 1, if it is the diagonal is occupied otherwise take the next element c and compare if col is equal to c $\pm$ 2 etc until all elements are tested.

The functions can now be defined as

```
(DE STARTBOARD NIL NIL)
(DE PRINTBOARD NIL (PRINT BOARD))
(DE TESTBOARD NIL
    (PROG ((B BOARD) (V 1) P)
          TOP
          (COND ((NULL B) (* SAFE PLACE FOR THE NEW QUEEN) (RETURN T)))
          (SETQ P (CAR B))
          (COND ((OR (EQ COL P)
                     (EQ COL (IPLUS P V))
                     (EQ COL (IDIFFERENCE P V)))
                 (* THE NEW QUEEN CAN NOT BE PLACED HERE)
                 (RETURN NIL))) (Soluions, Section 25)
```

```
        (SETQ B (CDR B))
        (SETQ V (ADD1 V))
        (GO TOP)))
    (DE PLACEBOARD NIL
        (CONS COL BOARD ))
```

An 8x8 array could naturally have been used as the representation of the board. A problem will then occur at backtracking and when an already placed queen must be removed from the array. The array is a global structure and elements set in the array since the last choice-statement must get their old values when a failure statement is executed. After backtracking to a choice-statement we want the environment to be exactly the same as it was the earlier time we were at this statement. This can be solved if a function with side effects (rplaca, setq, put, seta etc) saves its old value. This technique is used by INTERLISP in the editor and the history package (not included in INTERLISP/360-370). All functions with side effects have a /-form (/rplaca, /setq, /put, /seta etc) and these functions save a form every time they execute. If this saved form is evaluated the side effect will be undone.

Section 26

1. First printstat.

```
(DE PRINTSTAT NIL
    (TERPRI)
    <MAPC STATFNS (FUNCTION (LAMBDA (FNV)
                        (PRIN1 (CAR FNV))
                        (SPACES (IDIFFERENCE 20 (POSITION)))
                        (PRINT (CAR (CDR FNV>
    (TERPRI>
```

In unstat we unadvise and remove corresponding pair from statfns, by using repassoc (see example 7c in Section 11). If arg to unstat is NIL all functions are unadvised.

```
(DF UNSTAT FNS
   <COND ((NULL FNS) (SETQ FNS (MAPCAR STATFNS (FUNCTION CAR)))
                     (SETQ STATFNS NIL))
        (T (MAPC FNS (FUNCTION (LAMBDA (X)
                     (SETQ STATFNS (REPASSOC STATFNS X>
   (APPLY 'UNADVISE FNS)
   FNS>
```

In restat we only need to initialize the counters to 0 again.

```
(DF RESTAT FNS
   <COND ((NULL FNS) (SETQ FNS (MAPCAR STATFNS (FUNCTION CAR>
   (MAPCAR FNS (FUNCTION (LAMBDA (FN)
                  (PROG (TEMP)
                        (SETQ TEMP (SASSOC FN STATFNS))
                        (COND ((NULL TEMP)
                                  (* IN CASE WE RESTAT FUNCTION NOT
                                     ON STATFNS)
                                  NIL)
                              (T (RPLACA (CDR TEMP) 0)))
                        (RETURN (CAR TEMP>
```

Yes, (CAR IN FOO) can be used and stat will create the counter

```
%(CAR% IN% FOO%)
```

but we must redefine repassoc to make an equal-test instead of an eq-test.

Section 27

1.
```
(DE PPDUMP (POS1 POS2)
   (PROG (M N SLOT)
         (:: MAKE STACK POSITIONS IF THEY ARE GIVEN AS LITERAL ATOMS)
         (COND ((NULL POS1) SETQ POS1 (STKNTH -1 'PPDUMP)))
               ((LITATOM POS1) (SETQ POS1 (STKPOS POS1 -1))))
         (COND ((NULL POS2) (SETQ POS2 (STKNTH 4)))
               ((LITATOM POS2) (SETQ POS2 (STKPOS POS2 -1)))
         ((NUMBERP POS2) (SETQ POS2 (STKNTH (MINUS POS2) POS1))))
         LOP
         (:: OUTER LOOP FOR EACH FUNCTION BLOCK)
         (SETQ N (STKNARGS POS1))
         (SETQ M 1)
         LP
         (:: INNER LOOP FOR EACH SLOT IN A FUNCTION BLOCK)
         (COND ((GREATERP M N) (GO NXTPOS)))
         (SETQ SLOT (STKARG M POS))
         (PRIN1 (CDR SLOT))
         (SPACES 2)
         (PRINT (CAR SLOT))
         (SETQ M (ADD1 M))
         (GO LP)
         NXTPOS
         (COND ((EQ POS1 POS2) (RETURN)))
         (SETQ POS1 (STKNTH -1 POS1))
         (GO LOP)))
```

2.
```
(DE ::RESET NIL (RETFROM 'EVALLOOP))
```

Section 28

1. The value returned is -3 and the variable h in the prog will be bound to

   (FUNARG F [(A . 2)])

   If the call to function in g is changed the value 3 is returned.

2. We want dp to make a function definition to randomprocess to look like

   (LAMBDA (RANDNR) (FUNCTION (LIM) ++++++ (RANDNR)))

   To perform this dp is defined as

```
(DF DP (PROC LOCVAR PARAM . FNBODIES)
   (PUTD PROC
         (LIST 'LAMBDA
               LOCVAR
               (LIST 'FUNCTION (CONS 'LAMBDA (CONS PARAM FNBODIES))
                                LOCVAR))))
```

   spit is defined as

```
(DP SPIT (L) NIL (PROG1 (CAR L) (SETQ L (CDR L))))
```

   alternate is defined as

```
(DP ALTERNATE (PROCA PROCB) NIL
       (PROG (TEMP) (SETQ TEMP PROCA)
                    (SETQ PROCA PROCB)
                    (SETQ PROCB TEMP)
                    (RETURN (APPLY PROCB))))
```

   we create a process by

```
(SETQ ALTPROC (ALTERNATE (SPLIT '(A B C D E F))
                         (SPLIT '(1 2 3 4 5 6 7>
```

   and the process is called by evaluating

```
(APPLY ALTPROC)
```

Index

APPENDIX

HOW TO RUN INTERLISP/360-370 ON YOUR COMPUTER

This appendix should contain all necessary information depending on the installation for running INTERLISP/360-370. This page is only a skeleton of what information this appendix should contain and, hopefully, every installation will make an appendix of their own.

INSTALLATION:

COMPUTER:

$\left.\begin{array}{l}\text{OPERATING SYSTEM}\\\text{TIME SHARING SYSTEM}\end{array}\right\}:$

HOW TO RUN LISP:

Commands or procedures for running LISP.

FILEHANDLING:

How to allocate a dataset for LISP file handling.

How to delete a dataset or a memeber of a dataset.

How to list names of the LISP files (members of the dataset).

How to compress a dataset.

THE SAVE-FACILITY:

How to create a save. Allocation of dataset etc.

CHARACTER SETS:

Differences between the characterset used in LISP-details and the set used on your computer.

Differences between character sets on available equipment, such as terminals, line printers, card readers etc.

Special characters, such as
end of line
delete character
delete a line
control characters (attention).