ANTWERP PAPERS IN LINGUISTICS

nr. 9, 1976

# THE FORLI.OLB PACKAGE FOR LIST PROCESSING IN FORTRAN IV -- A USER'S MANUAL .

Luc Steels

UNIVERSITEIT ANTWERPEN

Universitaire Instelling Antwerpen

Departementen GER. en ROM. , Afdeling Linguïstiek.

Universiteitsplein,1, B-2610 Wilrijk-Antwerpen.

ABSTRACT

The paper discusses a number of functions and subroutines which all deal with list processing and are written in the programming language FORTRAN IV.

After a brief introduction to the concept of list structures, a user's manual is given.

CONTENTS

Preface

1. List Structures
1.1. Lists and atoms
1.2. The representation problem
1.3. Trees

2. List Processing in FORTRAN IV
2.1. Representation
2.2. Initialization
2.3. I/O routines
2.4. Processing functions

3. Operating systems

Conclusions

## Preface

Good tools are a cornerstone in the development and functioning of any (exact) science, and great care is normally taken in their design and use. Apart from the machinery involved in acoustic phonetics and related areas, in the language sciences there is only one basic instrument and that is a digital computer. However, digital computers are machines designed for a general purpose, hence in order to make them more suitable for a particular task, it is necessary to develop some additional components and add them to the machine. These components are normally compilers or interpreter systems which accept a particular programming language designed for a particular sort of problems. Our work is less ambituous than the design of a new programming language, although it serves the same purpose.

We started from the given fact that there was only a FORTRAN compiler available at the computer we were suppose to use and that for this machine (a PDP 11) and the currently implemented operating system (RSX ) the desired software (e.g. a LISP compiler or interpreter system) could not immediately be obtained. To fill the gap a project was started concentrating on the development of software for linguistic applications. This necessarily includes (i) list processing, (ii) recursive programming, (iii) string manipulations, (iv) flexibility in definition of functions (for work on semantics).

It was decided that the capability of doing list processing was the first step. For this purpose a library of functions and subroutines was created. A first version of the library has been used extensively by many persons over the last few months.It was felt to be a handy tool in coping with linguistic problems and a valuable help for people working on small machines with limited supporting software. In the second improved version, which is documented in this paper, some new features (e.g. more flexible I/O) are added, also new routines are added and some parts (such as dot-notation of list structures) were removed because they were not being used at all.

This paper is conceived as a user's manual of the library. The first section gives a brief introduction to list structures. For a more extended introduction and useful exercises we refer to the text books on LISP (e.g. Weissman, 1965). The second section explains all the functions and subroutines and how they should be used.

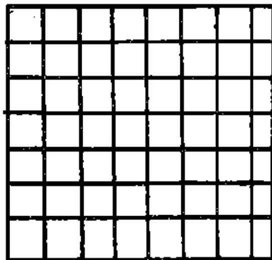
As this paper is meant as a reference text for people actually working with the library, we assume throughout the text that the reader is familiar with programming and has a working knowledge of FORTRAN IV (e.g. as covered by McCracken, 1965). An interesting and very helpful textbook during the implementation of the library was Waite's book (Waite,1973) on the implementation of software for non-numeric applications in general.

# 1. List structures

## 1. Lists and atoms

A data structure is (i) a set of cells, which can contain a certain datum, and (ii) a relation among the cells: a way of organizing them.
Some data structures are e.g.

a table:

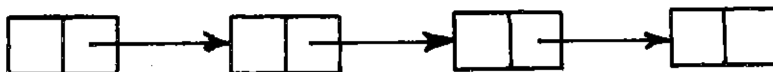


or

a linear array:

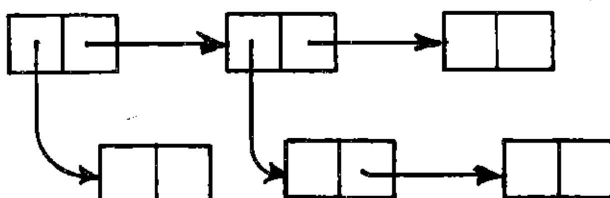
In these two data structures the location of the different cells of the data structure is defined in an implicit way, namely on the basis of the horizontal or vertical order. We can retrieve a value in one of the cells by addressing the position the cell takes in the data structure.

Suppose now that we make the structure explicit by drawing arrows if two cells are linked with each other.
E.g.:
(a)



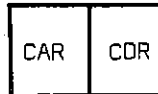
or

(b)

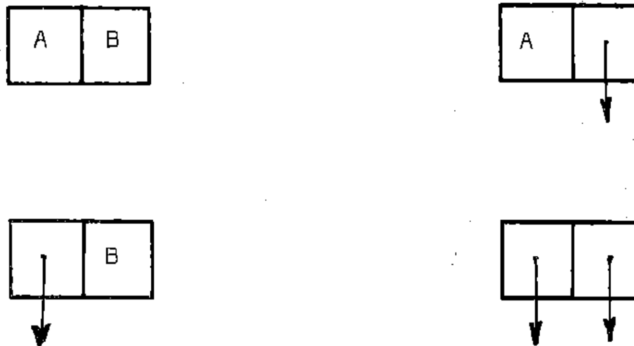


A data structure where the relations between the cells are made explicit by drawing links between them is a list structure. To locate a data cell in the structure we 'walk' through it till we come to the desired place.

From the examples it could be seen that in an explicitly linked data structure a cell (or box) contains two parts. These two parts are known as the CAR and the CDR (pronouns 'cudder') of the cell:




A CDR- or CAR-field contains either a data item or another pointer, i.e. a link to another cell.
Compare:

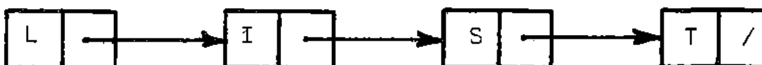



From now on we will call a datum an ATOM; it is considered to be a nondivisible entity. The second category in the discussion is the list, being a number of cells linked onto each other by their respective pointers.
E.g.:

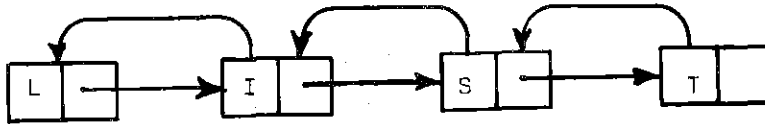
Note the slash at the end, it denotes the end of the list. Another name for the slash is NIL, denoting that there is nothing in that part of the cell. If a list contains no elements at all then it is also represented as NIL. In this case NIL is called the null list. So, if NIL is placed in a CAR- or CDR-field then we may assume that a list without any elements is attached to this field.

Some more definitions; consider:

(a)



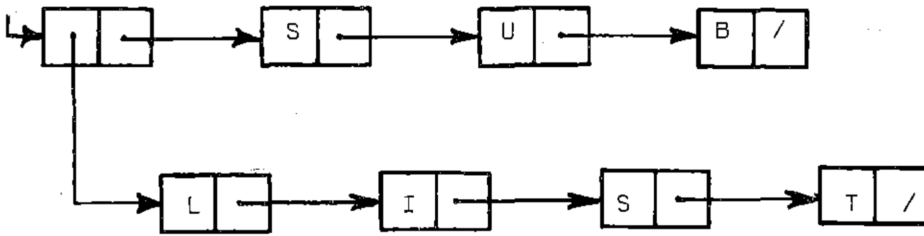
(b)




In (a) we have linked one cell to its successor, this is a one way list.
In (b) we have a link from one cell both to its successor and its predecessor, a two way list. From now on we will only deal with one way lists.

If a list contains a pointer in one of its CAR-fields, then the list starting from such a CAR-field is a sublist.

E.g.:




A list with sublists is called a branched list, a list without sublists is known as a linear list.

## 2. The representation problem

To have a successful data structure it is not sufficient to have a graphical representation. One must be able to write down the graphical representation in a linear way, i.e. algebraically. For tables and vectors, we do this by naming the whole data structure with a symbol (say X), and the different cells of the data structure are addressed by subscripts, e.g. X(1,2) denotes the first cell of the second column in a table called X.

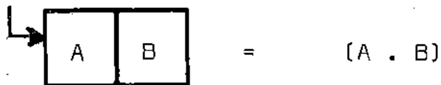
For list structures the solution to the representation problem is not so easy, simply because cells cannot be addressed by subscripts on the basis of their location , i.e. by referring to lines and columns. The problem is solved by the introduction of S-expressions with two particular formats: dot-notation and list-notation.

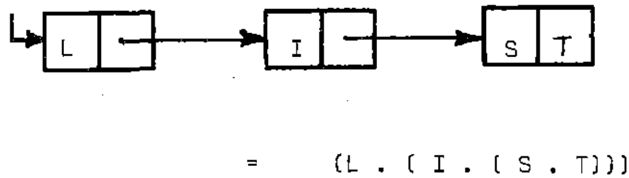
DOT-NOTATION

The dot-notation of a list structure is a direct mirror of its graphical representation. For each cell we introduce two brackets and one dot :

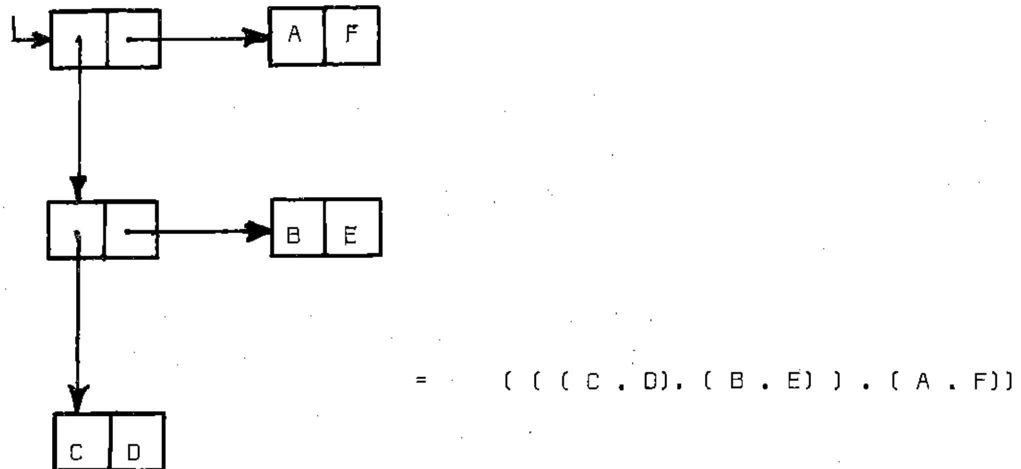
On the right side of the dot we write the CAR and on the left side the CDR. If the CAR or CDR contains a pointer, then we replace this pointer by the whole sublist depending from this pointer written in dot-notation.

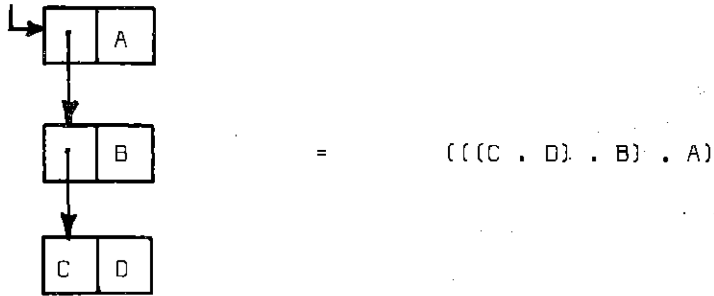
Examples:




= (L . ( I . ( S . T)))





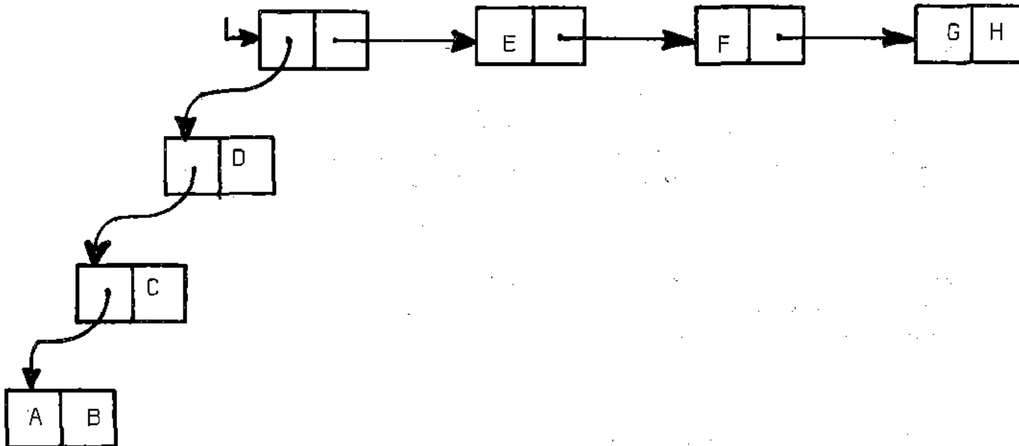

The following strategy can be followed for the construction of dot-notations from graphical structures.

(1) Consider a list to be linear and whenever a pointer appears, introduce a variable name for the sublist depending from this pointer.

(2) Similarly construct for each sublist on the same basis a linear list with variables when necessary.

(3) Replace all variables by their respective dot-representations.

E.g.:




F1 = ( A . B)

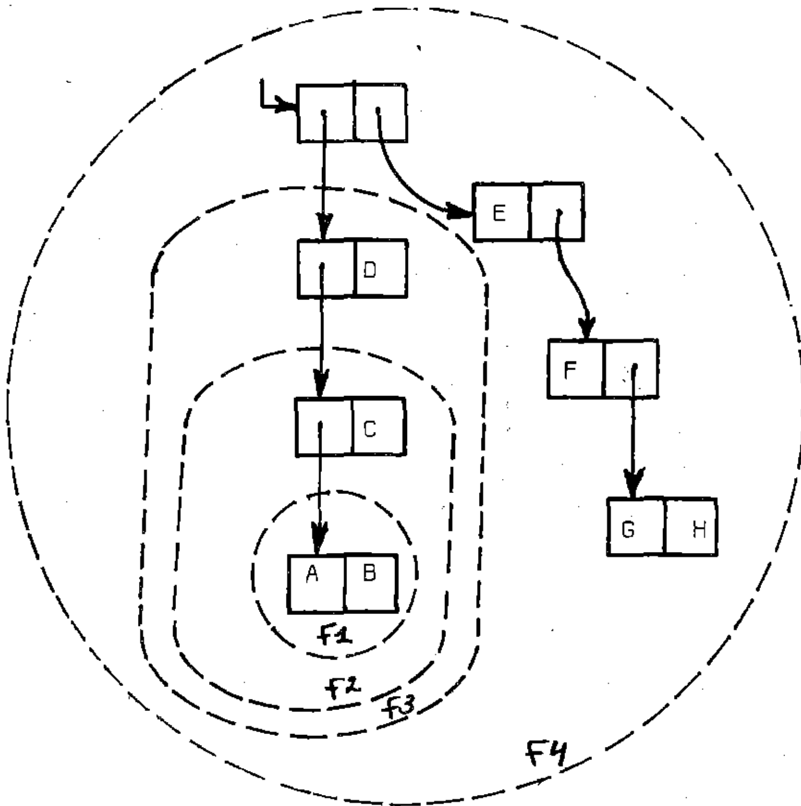
F2 = (( A . B ) . C )

F3 = ( ( ( A . B ) . C) . D )

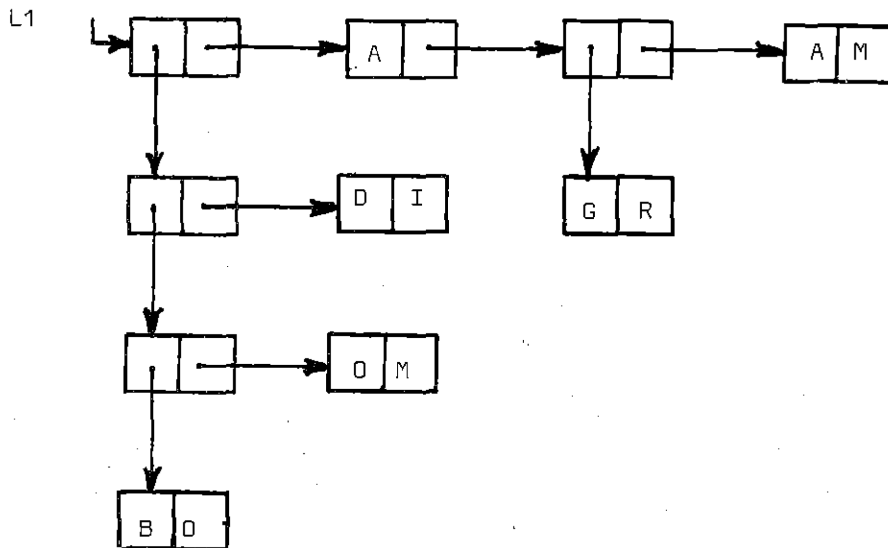
F4 = ( F 3 . ( E . ( F . ( G . H ) ) ) )

final result:

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




E.G. given:




We have the following sublists

L1 = (L2 . (A . (L5 . (A . M))))

L2 = (L3 . (D . I ))

L3 = (L4 . (O. M ))

L4 = (B . O)

L5 = (G . R )

Replacing all the variables yields:

L3 = ( ( B . O ) . ( O . M ))    (L4 in L3)

L2 = ( ( ( B . O ) . ( O . M ) ) . ( D . I ) )    (L3 in L2)

L1 = ( ( ( ( B . O ) . ( O . M ) ) . ( D . I ) ) . ( A . (L5 . ( A . M ) ) )

Finally:

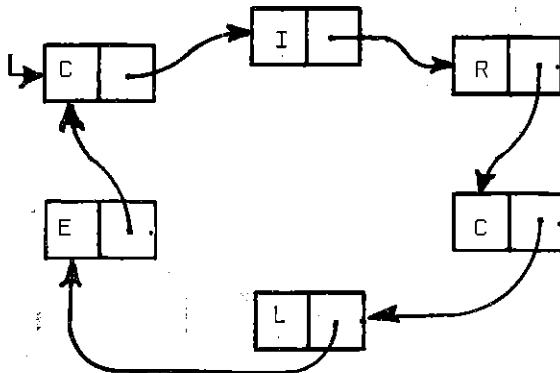
L1 = ( ( ( ( B . O ) . ( O . M ) ) . ( D . I ) ) . ( A .

( ( G . R ) . ( A . M ) ) ) )

(L5 in L1)

Graphically we had the following sublists:

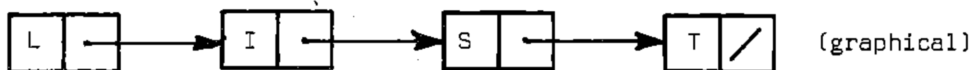
Although dot-notation is an immediate reflection of a graphical structure, there is already one sort of list structures that cannot be expressed namely a circular list. A circular list is a list where a pointer in some field points to a previous cell of the list.
Example:




Clearly a dot-notation of this graph would never come to an end.
The fact that circular lists cannot be expressed is however seldom felt as a drawback, certainly not in linguistic practice.

LIST-NOTATION

Although the dot-notation of lists is a very nice way of writing graphs into a linear format, it soon becomes extraordinary complex when the list structures themselves grow. Therefore another representation has been designed: list-notation. This goes as follows:
(i) A linear list is transferred by writing all the elements of the respective CAR-fields right after each other
E.g.:



(graphical)

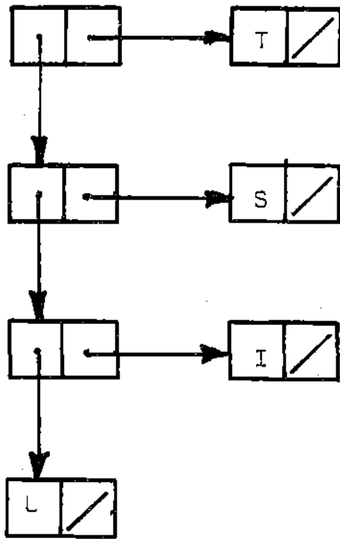
or (L . ( I . ( S . ( T . NIL)))) (dot-notation)

or (L I S T ) (list-notation)

(note that nothing is provided if there is an atom in the CDR-field)

(ii) As for dot-notation, as soon as there appears a pointer to a sublist in one of the CAR-fields, construct the list-notation for this sublist and replace the pointer by this sublist.

Example:




in dot-notation:

( ( ( ( L . NIL) . ( I . NIL)) .
(S . NIL)) . ( T . NIL))

in list-notation

( ( ( ( L ) I ) S ) T )

The technique for constructing dot-notation from graphical structures can also be used here to construct list-notation from graphical structures.

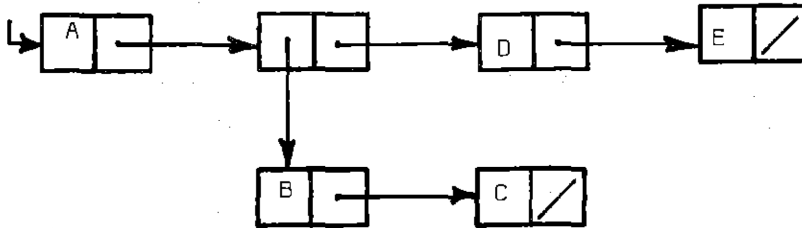
(i) Consider a list to be linear and whenever a pointer appears, introduce a variable name for the sublist depending from this pointer.

(ii) Similarly construct for each sublist on the same basis a linear list with variables when necessary.

(iii) Replace all variables by their respective list representations.

Example:

(i)



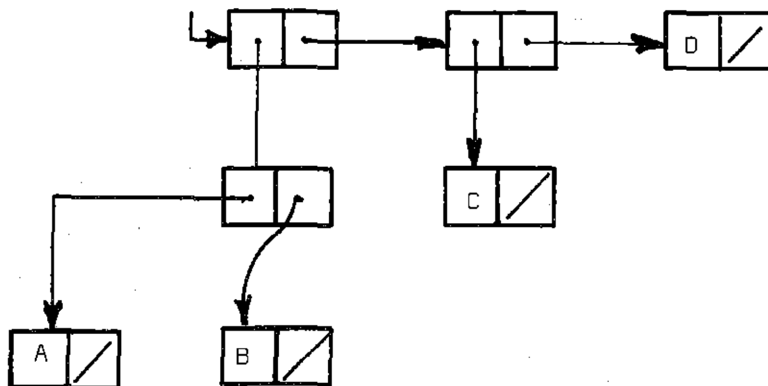

Let $L_o$ = (A L1 D E)

L1 = (B C )

then L1 in $L_o$ yields :

(A ( B C ) D E )

(ii)




In dot-notation: (((A . NIL) . ( B . NIL)) . ((( C . NIL) . ( D . NIL))))

We have the following steps:

L5 = (L1 L2 D)

L1 = (L3 B)

L2 = (C )

L3 = (A )

Finally L5 = ( ( ( A ) B ) ( C ) D )

Restrictions on list-notation:

(a) It is not possible to represent circular lists.

(b) Whenever an atom appears in a CDR-field we have to use dot-notation.

To see more clearly the relation between dot-and list-notation and the graphical representation we discuss the reverse way: from dot- or list-notation back to a graphical representation.

For this purpose, we can use the reversal of the previously used strategy:

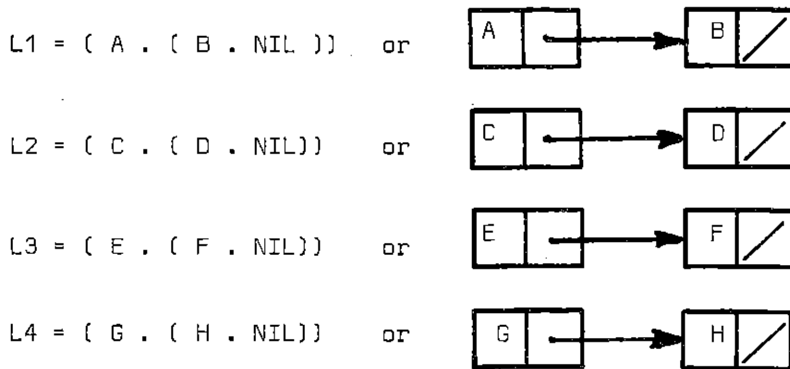
(i) Try to discover linear lists and give them a name

(ii) Construct the graphical representation for each linear list

(iii) Reconstruct the whole by replacing all the variables by their graphical representations.

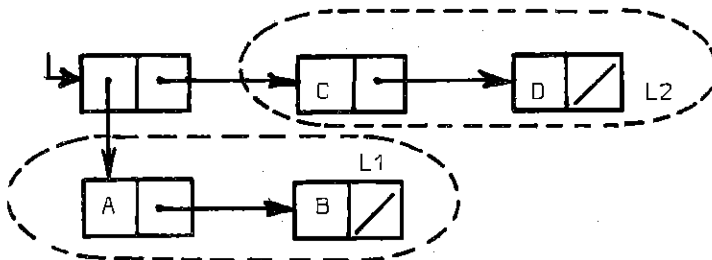
Example:

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

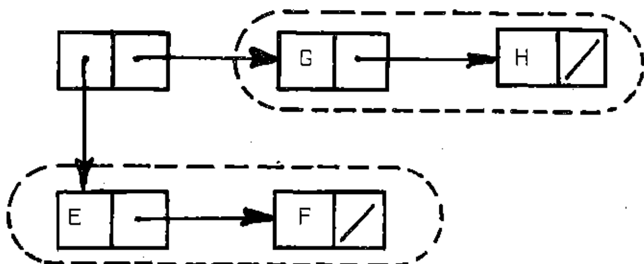



Construct L5 from L1 and L2

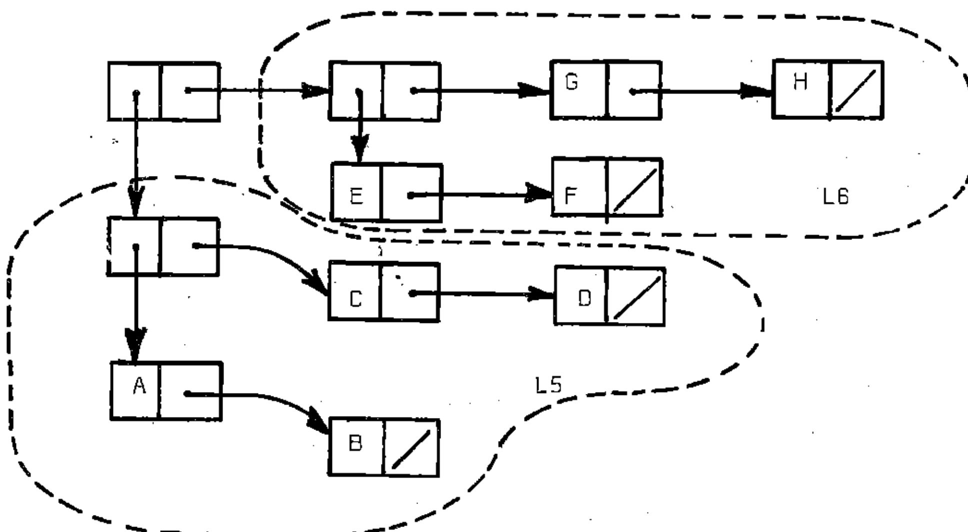
L5 = ( ( A . ( B . NIL) ) . ( C . ( D . NIL ) ) ) or



L6 from L3 and L4; L6 = ( ( E . ( F . NIL)) . ( G . ( H . NIL ) ) )




L5 and L6 yield the complete structure:



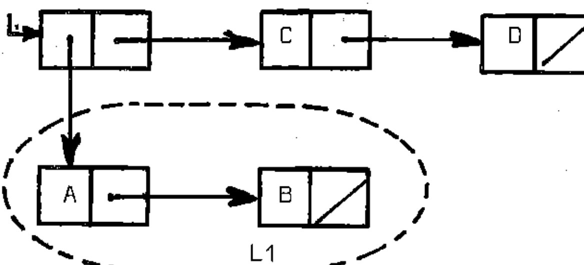

In the same way we construct graphical representations from S-expressions in list-notation. Example: ( ( ( A B ) C D ) ( E F ) G H )

L1

L2

L3 = (L1 C D )

or



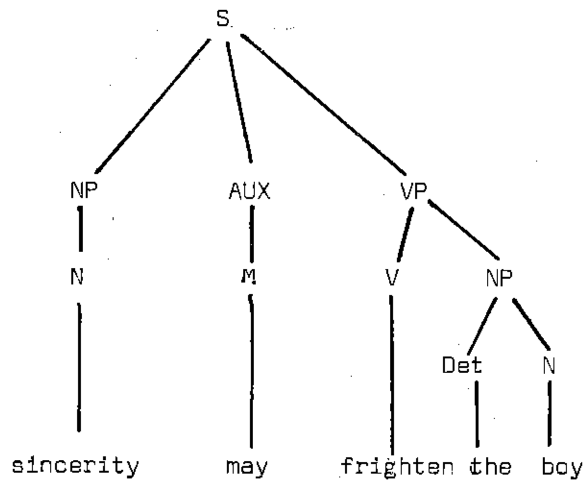
Note: The reader who is not yet thoroughly familiar with list representations should at this point consult a textbook and make more exercises on the translation from one representation into another one.

## 3. Trees

List structures are widely used in any sort of linguistic applications. There are two reasons for this (1) easy input and output of strings, and easy processing of alphanumeric data, and (2) (and this is even more important) easy representation of structures.

This last point will be illustrated more explicitly in this section. We will discuss the standard means of representing tree structures as a special case of list structures.

A typical representation used in linguistics is a labelled plane rooted graph or tree.




An alternative linear representation of the same information is the so called labelled bracketing:

$$(\ (\ (\ \text{SINCERITY}\ )_{N}\ )_{NP}\quad (\ (\ \text{MAY})_{M}\ )_{AUX}$$

$$(\ (\ \text{FRIGHTEN}\ )_{V}\ (\ (\ \text{THE}\ )_{DT}\ (\ \text{BOY})_{N}\ )_{NP}\ )_{VP}\ \ )_{S}$$

or

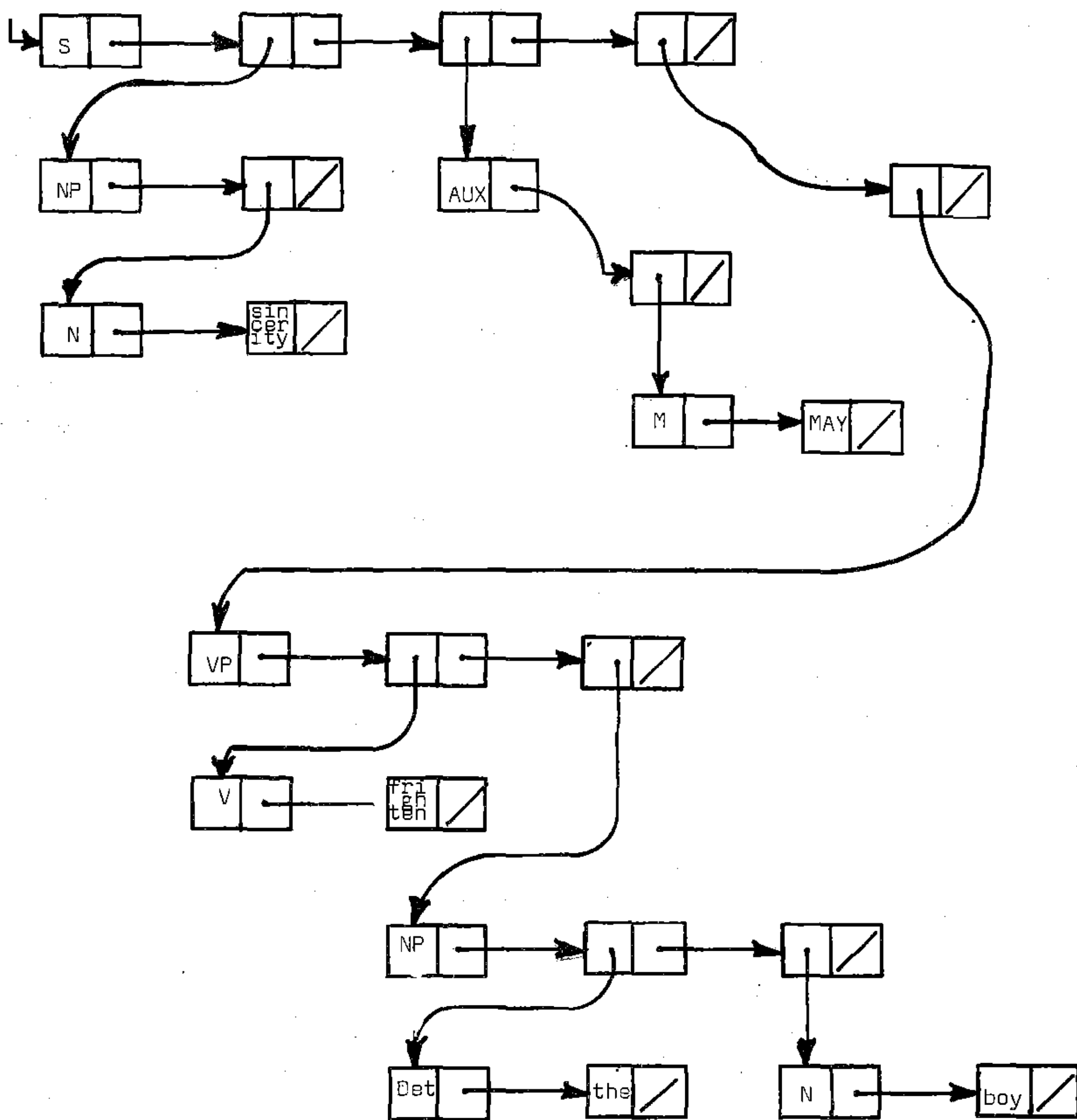
$$(_{S}(_{NP}\ (_{N}\text{SINCERITY})\ )\ (_{AUX}\ (_{M}\ \text{MAY})\ )$$

$$(_{VP}(_{V}\ \text{FRIGHTEN})\ (_{NP}\ (_{DET}\ \text{THE})\ (_{N}\ \text{BOY})\ )\ )\ )$$

respectively called right labelled bracketing and left labelled bracketing.
Now, if we take the left labelled bracketing and write all symbols on one line we obtain:

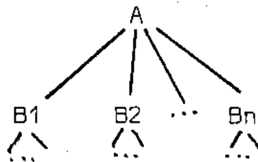
(S (NP (N SINCERITY) ) (AUX ( M MAY) ) (VP (V FRIGHTEN) (NP ( Det THE) (N BOY))))

and this is nothing else but the list-notation of a list structure.
The graphical representation of this is:


S
NP
N
sin cer ity
AUX
M
MAY
VP
V
fri gh ten
NP
Det
the
N
boy

Because of the importance of the relation between trees in graphical and list-notation we define explicitly the relationships between the two.

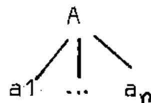
(i) Given a tree structure




with A, B1, ... , Bn nonterminal nodes, then the equivalent list-notation is

(A (B1 ... ) (B2 ... )...( Bn ... ) )

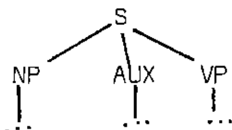
(ii) Given a tree structure



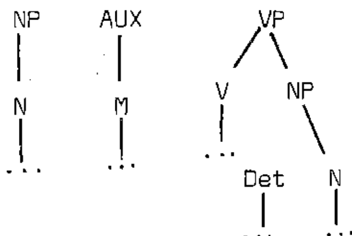

with A a nonterminal node and a1, ... , $a_n$ a terminal node, then the equivalent list-notation is

(A a1 ··· an)

Example:

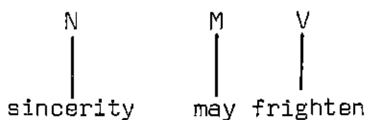



(S (NP ... ) ( AUX ... ) ( VP ... ) )

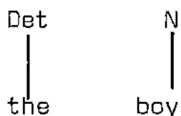



(S (NP ( N ... )) ( AUX ( M ... ) )
(VP ( V ... ) ( NP ( Det ... )
( N ... ) ) ) )

finally:




(S (NP ( N sincerity) ) ( AUX ( M may))
(VP ( V may )(NP ( Det the ) ( N boy ) ) ) )



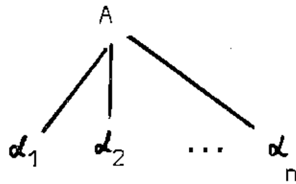
Reverse:

Given a list

(A $\alpha_1$ $\alpha_2$ ...$\alpha_n$)

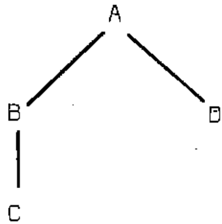
with $_1$ ... $_n$ sublists or atoms

then the equivalent tree is




Example:

Given (A ( B C ) D ) the equivalent tree is



## 2. List Processing in FORTRAN IV

Fortran IV is not a list processing language, that is a language where lists are available as data structures. We will now discuss a number of subroutines that make it nevertheless possible to work with list structures in FORTRAN. These functions and subroutines deal with the following aspects:

(i) initialization

(ii) input/output

(iii) list processing.

Before discussing the use and functioning of all these subroutines, we give some preliminary remarks on the nature of the representation being used.

### 2.1. Representation

For the machine representation of list structures, we take a data structure already available in FORTRAN: the integer declared 2-dimensional array. Then we let each cell in the graph representation correspond to a row in the table and the three parts of the cell: the AF-field, the CAR-field and the CDR-field are the first, second and third column of the table respectively. So, we can address a part of a cell by giving a row and a column.

This representation may not be the most compact one possible, it is undoubtely the best solution as regards the transportibility to other machine configurations.

Strings of characters (e.g. the name of an atom) are represented by coding them into integers and storing these in the same table.

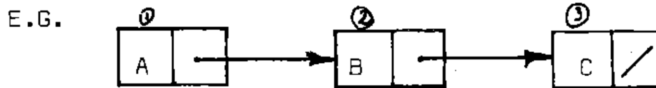
The user , however, does not need to worry about the actual representation and the detailed mechanics of the storage and processing of list structures. He can 'think' in terms of list structures and atoms instead of integers and table representations. In this way it becomes possible to write interesting programs,using list processing, in FORTRAN without bothering about the complexity of the details.

As the graphical 'cells' correspond to rows in the table, it follows that pointers to lists are actually integers (denoting the appropriate row). And (in the FORTRAN program) the name of the list is a variable with as value the row in the table. Again, the user does not (and he does not need to) know what the actual values are.

Note that

1. A pointer to a list is always a pointer to the first cell of the whole list. But the routines (e.g. for output) consider this a pointer to the whole list.

E.G.




If we call this list (in our Fortran program) I, than the value of I will be 1 (according to our supposed labelling).

2. A pointer to an atom is a pointer to the base cell of the atom.

3. In the discussion, it is handy to label the cells in the list structures in order to talk about them. These labels are not necessarily those used in the actual processing.

4. NIL (the null list) has the value zero. That means that if Ø is assigned to the variable NIL in a program, we can refer to the nullist with NIL.

We now start the discussion of the functions and subroutines that make it possible to use list structures in FORTRAN.

## 2.2. Initialization

- As integer type data are used all the time, it is useful to start any program or subroutine as follows:

```
IMPLICIT INTEGER (A-W)
```

- In a list processing system there is normally a so called free list created at the start. When in need of a piece of list structured memory, one takes 'cells' from this free list and when these cells are no longer needed they are returned to the freelist.
The creation of this freelist is the task of a special subroutine INIT, after the subroutine is called, the system is ready to start.

## 2.3. Input/output routines

(i) INPUT

RLIST is an integer function with 3 parameters:

I1 a pointer to the position where the reading should start

I2 a pointer which results in the final position after execution of the function

I3 a code for the device from which the system should read.

The result of RLIST is that all decoding and storing is performed and that a pointer to a list (or atom) is returned as result.

The following conventions hold for the arguments:

1. If I1 is equal to Ø, than a new line of input is consumed but the line is NOT printed out during reading

If I1 is equal to 1, a new line of input is consumed and this line is printed on the default output device.

If I1 is greater than 1, the system starts reading on the latest consumed line.

Whenever a line is completely processed, but more characters are needed, the system keeps reading new lines from the input device until a complete list (or atom) is found.

2. I2 is set to the final character used in the RLIST-process. So, with I2 we can keep on reading on the same line if we take this as starting point for the next call of RLIST.

3. There are 4 devices normally connected to the system.

(i) by default: CR: (card reader) with logical unit number 1
LP: (line printer)with LUN 6
(can be both connected to teletype during taskbuilding)

(ii) two files on disk: FORØØ4.DAT and FORØØ5.DAT , i.e. LUN 4 and 5 respectively

Now, if I3 = Ø, then the input device is the card reader.
if I3 = 1, the input device is the FORØØ4.DAT file on disk
if I3 = 2, the input device is the FORØØ5.DAT file on disk

Examples:

(1) I1 = RLIST (1,I,Ø)

A list (or atom) is read from the card reader, the input lines are printed.

(2) I1 = RLIST (Ø , I, 1)
I2 = RLIST (I, I , 1)

Two lists (or atoms) are read from a file FORØØ4.DAT, the second list (or atom) comes after the first one, possibly on the same line. The line is not printed.

Notes:

1. Blanks are ignored if not meaningful.

2. Superfluous right brackets on the last card are ignored. (but if you keep reading on the same line, an error message will follow: 'TOO MANY RIGHT PARENTHESES'.

3. A lack of right brackets will make the system look for further brackets and therefore consume the rest of input cards. Then a message will be issued: 'TOO MANY LEFT PARENTHESES'.
So, a lack of right brackets is a fatal error, in that it is noticed only when all cards have been read.

4. the null string can be represented in the input by NIL and ().
NIL is the only atom that is present as soon as the program starts.
(The integer value of NIL is Ø.)

5. Each character that is given as input is coded directly into an integer. We list here all the characters and their codings, other characters are not accepted by the input and output subroutines.
If they are given as input, a message 'UNRECOGNIZED CHARACTER' will be issued.

| CHAR. | CODE | CHAR. | CODE | CHAR. | CODE |
|---|---|---|---|---|---|
| ƀ | 1 | M | 17 | 7 | 37 |
| A | 2 | N | 18 | 8 | 38 |
| E | 3 | P | 19 | 9 | 39 |
| I | 4 | Q | 20 | . | 40 |
| O | 5 | R | 21 | , | 41 |
| U | 6 | S | 22 | * | 42 |
| Y | 7 | T | 23 | / | 43 |
| B | 8 | V | 24 | - | 44 |
| C | 9 | W | 25 | + | 45 |
| D | 10 | X | 26 | _ | 46 |
| F | 11 | Z | 27 |  | 47 |
| G | 12 | ( | 28 | : | 48 |
| H | 13 | ) | 29 | ; | 49 |
| J | 14 | Ø | 30 | ? | 50 |
| K | 15 | 1 | 31 | " | 51 |
| L | 16 | 2 | 32 | = | 52 |
|  |  | 3 | 33 | ) | 53 |
|  |  | 4 | 34 |  |  |
|  |  | 5 | 35 |  |  |
|  |  | 6 | 36 |  |  |

6. An important (but difficult) question is the fact that there is a fundamental distinction between the FORTRAN program and the variables for lists and atoms used therein and the users' specification for the atoms and lists , a distinction which is not so stringent in LISP e.g. , due to the QUOTE-feature.
Clearly the bridge between the two is the RLIST function. Therefore any atom that is used as an entity in the program should be read in by RLIST.
E.g. suppose 'NOUN' is an entity which is being referred to in the program, we can write

```
    NOUN = RLIST (1,I,1)
```

where NOUN is on the card. From then on the variable 'NOUN' (in the FORTRAN program) will refer to the same object as the atom NOUN in input/output.
7. A special sort of atom, called p-atoms, can be processed by RLIST. They are not stored in the initial atom dictionary , but on a special array (of 200 characters length). Also all information about p-atoms can be removed by one single operation (CLEAR).

(ii) OUTPUT

( a) PRLIST is a subroutine with three parameters I1, I2,and I3.

I1 is a pointer to a list (i.e. to the first element on a list)

I2 is an integer value denoting the position on the outputline from where the system should start printing;if I2 is Ø , a line is left open and the system starts from the first character on the next outputline.

I3 is the device on which the output must appear.

If I3 = Ø then the output appears on the device with LUN = 6, (the line printer by default)

if I3 = 1 then the output appears on the device with LUN = 4 (normally a file FORØØ4.DAT on a disk)

if I3 = 2 then the output appears on the device with LUN = 5 (normally a file FORØØ5.DAT on a disk)

The result of PRLIST is that the whole list structure pointed at by I1 is recoded in alphanumeric characters and transferred to the device.

NOTES:

1. PRLIST also handles atoms

2. If list notation is impossible, dot notation is used, but only at the point where it is necessary:

e.g. given (A . (B . ( C . D ))) , this will be printed as

(A B C . D)

3. When I2 is greater than one, all characters before I2 on the output line are blanks. One can use this feature for editing.

e.g. Suppose you want the following as output:

THE NAME IS : JOHN , where'the name is:' is in the program and John an atom referred to by the variable <u>name</u>, then the output can be obtained by the following lines of FORTRAN.

```
      CALL PRLIST (NAME,14,Ø)
      WRITE (6,1)
    1 FORMAT (1H+,'THE NAME IS :')
```

(b) PLOTLI

A special program (called PLOT) has been implemented (in cooperation with P.Reypens) to plot tree structures on the plotter.

Tree structures are list structures with the following restriction

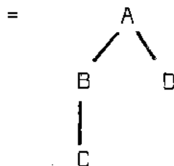
(i) the first element of the list is an atom (because this is the label)

(ii) the second element is either an atom or a list as specified in (1).

(iii) there should be at least two elements in a list.

e;g.: given (A ( B C ) D)

=




The list representation of a tree corresponds exactly to a right labelled bracketing of the string formed by its leaves.

The user can use the program PLOT (which is ready for running on disk), after first writing the lists on a special file by means of a subroutine called PLOTLI, and then running the program PLOT after his own job is finished.

PLOTLI

PLOTLI is a subroutine of 4 arguments: I1,I2,I3,I4.

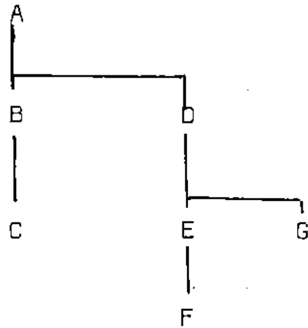
I1 is a pointer to the list

I2 denotes a value for the size of characters, of horizontal lines and the space between the leaves. This value is equal to I2 x 0,25 cm. So, if I2 is set to 1, the size of the characters will be 0,25 cm which is more or less the normal size.

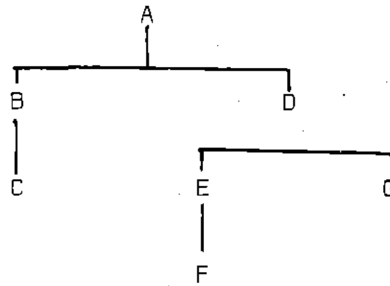
I3 denotes either 0 or 1. If I3 = 0 then the tree is not centered, if I2 = 1 the tree is centered, i.e. the lines from dominating nodes will end at the middle of the bar connecting the dominated nodes.

e.g.: given (A ( B C ) ( D ( E F ) G ) )

with I3 = 0




with I3 = 1




I4 denotes either 0 or 1.

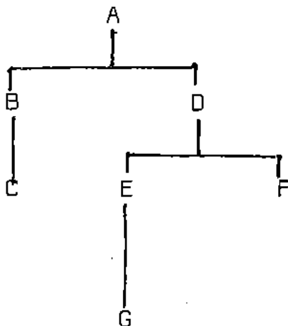
if I4 = 0 then the leaves will 'hang' right under their dominating nodes,

if I4 = 1 then the leaves are plotted on one line.

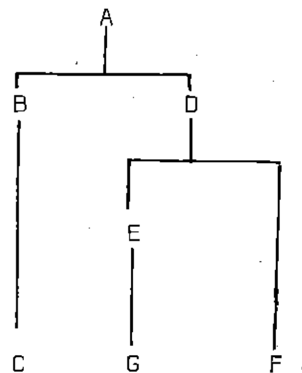
e.g.:

given the same list as in the above example:

with I4 = 0




with I4 = 1



Note:

1. Files from PLOTLI are written on FORØØ4.DAT , so, do not confuse this with other output on this file by PRLIST.

2. When all structures to be plotted are processed by PLOTLI, one should call the CLOSE subroutine in the fortran program, in particular CALL CLOSE (4). This is needed to 'close' the file, i.e. add an end of file symbol to it.

Now we proceed with the list processing functions and subroutines themselves.

2. . Processing functions

(a) Elementary

There are 6 elementary functions and subroutines, three which WRITE information in a field of a cell, and three which READ information from a field of a cell.

- <u>READ</u> (three integer functions)

CAR (I1)

where I1 is a pointer (address) of a cell in the memory. The result is the value of the CAR-field of the cell.

E.G.: Let I1 be (A B C) then CAR(I1) is a pointer to the atom A.

CDR(I1)

where I1 is a pointer to a cell in the memory. The result is the value of the CDR-field of the cell.

e;g.: Let I1 be (A B C ) then CDR(I1) is (B C)
Let I1 be (A ( B C )) then CDR(I1) is ((B C ))

AF(I1)

where I1 is a pointer to a cell in the memory. The result is the value of the AF-field of the cell.

e.g.: Let I1 be A then AF(I1) is 1
le I1 be (A) then AF(I1) is 0.

The AF contains either 1 or 0 and no other values.

- <u>WRITE</u> (three subroutines)

BECAR (I1,I2) (read 'be car of I1 I2')

This subroutine stores the value I2 in the CAR-field of the cell pointed at by I1.

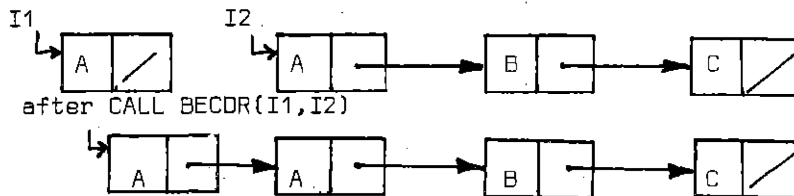
E.g. BECAR (I1,I2) with I1 (A B C) and I2 ((A)) then I1 becomes ((A) B C)

BECDR(I1;I2)    (read 'be cdr I1 I2')

this subroutine stores the value I2 in the CDR field of the cell pointed at by I1.

E.g.:   CALL BECDR (I1,I2) where I1 is (A) and I2 is (A B C ), results in ( A A B C )

i.e.:

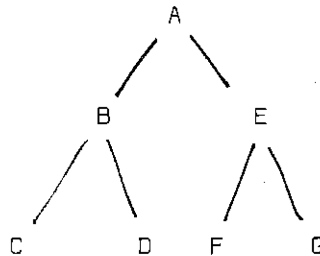



In view of the applications of tree structures, we add two special functions to it.

(i) SON (I1)

where I1 is a pointer to a 'node' in the tree. The result is the leftmost node depending from I1.

E.G.: Let I1 be (A ( B C D ) ( E F G )) as a tree




then the son of I1 (which is the top cell) is (B C D ) and the son of this is (C).

(ii) BRO (I1)

where I1 is a pointer to a 'node' in the tree. The result is a pointer to the first node y on the right of a node x, such that x and y are both depending from the same node z. When there is no such node, the result of BRO is nil.

e;g; Given the same tree as in the previous example, then BRO(SON(I1)) is (E F G )

Note:

As regards the elementary functions there is also BEAF(I1,I2) , read 'be AF of I1 I2', which stores the value I2 in the AF-field of the cell pointed at by I1. Unappropriate changing of the contents of the AF-fields can however cause serious trouble, especially for the output routines. A normal user does however not need to change the contents of AF-fields.

(b) Storage manipulation

There are two special operations to create a new cell and one to give back an already created cell.

NEW (I1)

is a subroutine that takes a new cell from the free list and sets the pointer I1 equal to this cell. It is a sort of initialization that takes place whenever we want to start a new list.

BACK (I1)

is a subroutine which is the reverse of NEW, it returns one single cell pointed at by I, back to the free list.

NEW and BACK are complementary in the sense that the execution of

```
CALL NEW (I1)
CALL BACK (I1)
```

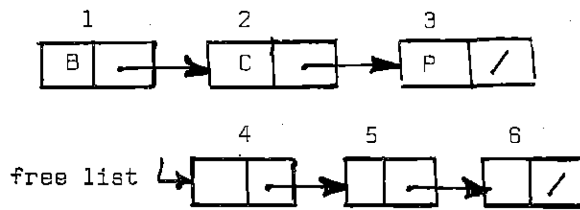
returns the whole memory structure as it was before the execution.

Another way of storage manipulation is the pushdownstore feature.

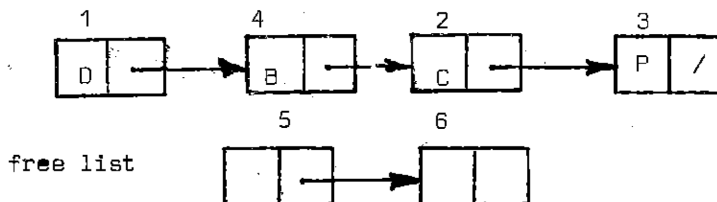
PUSH (K,I) (read ' push K on I')

is a subroutine where I is a pointer to the first cell of a list. It pushes the list I down, i.e. creates a new cell <u>on top</u> of I and stores K in the CAR-field of the (new) top of the list.

E.g.: suppose
With I = 1




then after CALL PUSH (L,I) we get with L is the atom D

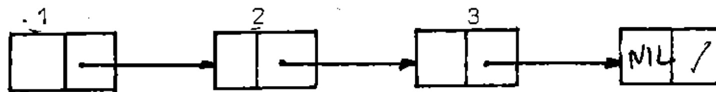



Note that the pointer to the head of the list does not change.

POPUP (K,I) ( read 'popup K from I')

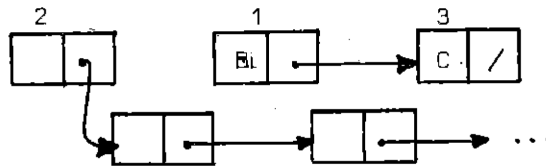
is a subroutine where I is again a pointer to the first cell of a list. The subroutine takes the contents of the first cell and sets it equal to K, then the first cell is removed from the list.

E.g. suppose (with I = 1)




then after CALL POPUP (K,I) we get

freelist




Note that I still points to the same cell, it is in fact the second cell that has been removed; K now points to the atom A.

Remarks:

1. It should be clear that a new element can only be placed on top of the pushdown store AFTER the pushing of the list and an element should be read from the top BEFORE the popping up operation is performed.

2. To use a list as pushdown store, it should first be initialized by calling NEW. When the last element is popped up from the list in a popup operation, the cell is automatically returned to the free list.

Other subroutines to remove information

1. When INIT is called at any point in a program, all previous information is lost and the memory contains only the freelist.

2. To remove all information as regards p-atoms, the routine CLEAR should be called

3. To remove a list structure without destroying the atoms, one should call the subroutine ERASE (I1) , where I1 is the address of the first cell in the list.

(c) Other list processing routines

(1) ADD (I2,I1) (read 'add I2 to I1)

is a subroutine with 2 parameters I1 is a list, and I2 is an atom or a linear list of atoms.
After execution, each atom of I2 is added to I1 <u>iff it is not present yet</u>.

e.g. Let I1 = ( A B C )
and I2 = ( C B A )
then after CALL ADD ( I2, I1)
I1 = (A B C)

Let I1 = ( A B C )
and I2 = ( D E F )
then after CALL ADD (I2,I1)
I1 = ( A B C D E F)

(2) ATTACH (I2,I1) (read 'attach I2 to I1')

is a subroutine with 2 parameters, I2 is a list
After execution a copy of all elements of all elements of I2 is added to I1

e.g. Let I1 = (A B C )
and I2 = ( C B A )
then after CALL ADD (I1,I2)
I1 = (A B C C B A )

Note that I2 is available for further processing afterwards.

(2) APPEND (I1,I2,I3)

is a subroutine that(i) creates a new cell,(ii) hangs it on the CDR of I1, (iii) puts I2 in the CAR of I1, and (iv) sets I3 equal to the new cell.
APPEND is especially useful in the construction of list structures because one can 'walk further' with a pointer while hanging new cells each time to the list.

(4) COPY (I1)

is an integer function that creates a completely new sequence of cells identical in structuring and with the same atoms on the same places as in the argument list I1. Copy returns as value a pointer to the newly made list.

(d) Manipulating the property list.

With each atom there corresponds a property list (for short p-list) on which the user can store pa .s of properties and values and consult afterwards whether a certain property is present and if it is present what property it is.

(i) PROP (I1,I2,I3)

is a subroutine which stores the property I2 with the value I3 on the property list of the atom I1.
I2 can be a linear list, I3 can be any sort of list or an atom.

(ii) GET (I1,I2,I3)

is a subroutine which checks whether the property I2 is on the p-list of the atom I1, if so I3 is set equal to the value of I2, if not the value of I3 is set equal to NIL.

If I2 is a list then I3 is returned iff all items on the original list which constitute the property are presented on I2. There may be more elements in I3, and the order is of no importance.

(e) Predicates

Finally some predicates are available to check whether the contents of the car-field of a cell is equal to an atom or a list.

To use this predicates, they should be initially declared to be of type LOGICAL.

(i) ATOM (I1)

this logical function checks whether the cell addressed by I1 is an atom or not. If it is an atom the result of the function is .TRUE. else the result is .FALSE.

(ii) LIST (I1)

this logical function checks whether the cell addressed by I1 is a list or not. If it is a list, the result is .TRUE., else .FALSE.

3. Operating systems (in relation to RSX)

The whole library of functions and subroutines is called FORLI.OLB;1 and is stored on disk [120.00] .
Suppose you have a main program called MAIN, which is in a compiled form stored on disk , say DK1:, and some routines in it are used from the library. Let us call the program which should result after taskbuilding PROG. Then the relevant instruction to the taskbuilder is:

TKB DK1: PROG = DK1: MAIN, DK1: BLOCK, DK1: FORLI.OLB;1/LB
The file DK1:BLOCK is a necessary special subroutine filling a commonzone with data.

Due to memory limitations, it is not possible to link the plot-routines directly to a program. Instead the relevant information is written on disk by PLOTLI in a file called DK1: FORØØ4.DAT and this file is used by another program, available on the [120.00] disk called PLOT.
Suppose there was a larger memory for taskbuilding available, then the plotting routines can be activated by calling PLOTL with the same paramèters as PLOTLI.

## CONCLUSIONS

The FORLI library discussed in this paper is used for various linguistic tasks such as parsing systems, question/answering systems, production systems, etc;. The library is constantly growing to cope with the problems we are dealing with. At the moment special subroutines are constructed for processing networks, e.g..
Although it is undoubtely more appropriate to use such a programming language as LISP, we are at the moment quite happy with the library. One of the advantages is that the programmer can manipulate the lists in a very powerful way, more powerful than in LISP, and avoid unefficient use of the store (think about garbage collection).

REFERENCES

Mc Cracken, D. (1965) A guide to FORTRAN IV Programming. Academic Press New York.

Waite, W. (1973) Implementing software for non-numeric applications. Englewood Cliffs.

Weissman, C. (1965) LISP 1.5. Primer. Belmont, California.

ISSUES OF THE ANTWERP PAPERS IN LINGUISTICS

1. - 1975 Luc Steels: Parsing systems for Regular and Context-free languages.

2. - 1975 Johan Van der Auwera: Semantic and Pragmatic Presupposition.

3. - 1975 Luc Steels: Completion grammars and their applications

4. - 1976 Georges de Schutter & Eddy Kockx: Meervoudsvorming en vervoeging in het Nederlands. Een morfofonologische proeve.

5. - 1976 Luc Steels & Dirk Vermeir: On the formal properties of completion grammars and their related automata.

6. - 1976 Luc Steels: Producing natural language from semantic information.

7. - 1976 Georges de Schutter: Een semantisch-syntak tische beschrijving van adjektieven in het Nederlands.

8. - 1976 Manuel Aguirre: Factuality and Modality.

Special Issue:

L. Steels (ed.) Advances in natural language processing. Preprints of a workshop held at the University of Antwerp (UIA), October 1976.