28

1965

"International Conference on Computational Linguistics"

MORPHISM: A SYMBOLIC LIST PROCESSING LANGUAGE

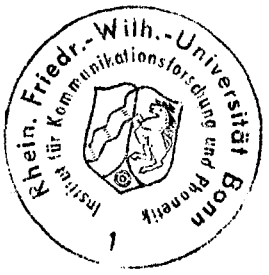
Vandenburgh Harold R.

Laboratoire de Calcul Automatique

Faculté des Sciences

2, Place Pasteur

RENNES - FRANCE



VANDENBURGH

MORPHISM: A SYMBOLIC LIST PROCESSING LANGUAGE

ABSTRACT

With the formal study of natural languages, we have obtained some residual results which greatly increase our power of logical manipulation. Coupled with a computer, our language extends enormously the external logic of our computing device. This paper is concerned with a well-defined symbolic list processing language and a set of operators for this language. These operators, which we call morphisms, are themselves operators thereby giving us an indefinite nesting ability. We have defined a schematic representation of our operators (called a comb-scheme) and have then proceeded to describe a numbering technique which allows us to use these operators with a computer. Since our operators and lists are defined at the time of execution, our ability to change them is unlimited.

The pattern of this paper is to build up from an alphabet to words, then non-connected words, then lists and the operations on lists and finally operations on the operations. We have taken, what we hope to be, descriptive examples in linguistics and non-linguistical applications in the hope of being representative of our language and easily understandable.

MORPHISM: A SYMBOLIC LIST PROCESSING LANGUAGE

Introduction

We at the Faculté des Sciences of the Université de Rennes are interested in the mathematical study of languages. Since we have a computer (1620), it is only natural that we attempt to combine the two. With languages being our major interest, we find that an extensive study of lists and symbolic list processing techniques will yield a most powerful and broad spectrum of results when applied to a computer. It appears that problems involving natural languages and certain types of mathematical problems are computer solvable when list processing techniques are applied to them. There are also linguistical problems which yield beneficial results when they are approached in this manner. We are therefore going to define: lists, operations on lists (which we call morphisms), and a comb-scheme which allows us to manipulate the lists and morphisms; this scheme gives us the ability to manipulate almost any pattern that can be handled in a logical manner (in linguistic usage, the term logical will refer to letters, words, sentences, phonemes; in a predetermined or random occurrence).

We have defined our operators in such a manner that we allow for an indefinite nesting; adding to a computing machine an external logical system. Also, our operators, in the computer, are defined at processing time so our ability to change them at execution time is unlimited.

This paper can be considered to be divided into four sections. Part one defines our terms and gives definitions for such things as an alphabet, lists, words, non-connected words, etc. Part two defines a morphism and our operators on lists. Part three shows our comb-scheme. Part four consists of examples: in linguistics; an example in the logical generation of a flow diagram into a computer program; and finally, an example of the solution of a mathematical problem (the arithmetic operations on polynominials of n variables).

1. Definitions

1.1 Words and Symbols

Since we are using a computer, let us define our alphabet to be composed of:

- the Latin capital letters
- the blank (#) for notational purposes but on printed page a blank.
- the punctuation marks . and ,
- the digits

excluding: = - $ / + @ ( ) and O which is also acceptable to the computer.

1.2 Non-connected words

Let us put in parallel the Latin phrase:

nec adulatoribus latus praebes

and its French equivalent:

et tu ne prêtes pas le flanc aux flatteurs;

The Latin word nec corresponds, in French, to the three non-consecutive words et ... ne ... pas and the Latin word praebes corresponds to tu ... prêtes

Thus it appears that for the words of a language, on the same level as words, there is a series of words not necessarily consecutive: the non-connected words. For a non-connected word, the components will be separated by the symbol: - . For example:

ATA - BRIQUE

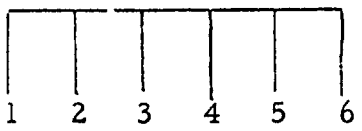
RI - DO -

are non-connected words. The first has two components and the second has three components.

1.3 Comb-scheme

In the formulas, as in algebra, we designate a variable by x (and eventually intervening variables by x_1, x_2...); a non-connected word of N components will be schematized by a comb or comb - scheme

(eventually transformed into a number) of N teeth ordered from left to right (eventually numbered from 1 to N) such that the first tooth represents the first component, the second tooth the second component, etc.



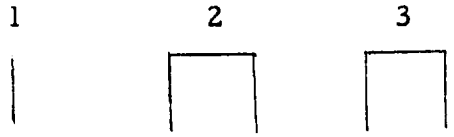
1.4 Lists of non-connected words;

We will separate the lists of non-connected words by a / . For example:

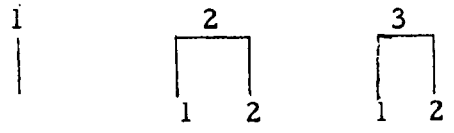
RA / TA - PLAN / PLAN - PLAN

is a list of three non-connected words: the first of one component; the other two, two. We will schematize the list by:

or if need be by:



or again by:



1.5 Designation of lists and their types

We will designate a list by a symbol (name, a set of letters and digits). We will enclose the symbol by the sign @ . This avoids

confusing the name of a list with elements of the list e. g. @AB5@ = A - B / C / D - E - F where AB5 is an element (or word of our alphabet); but @AB5@ designates a list. The list is:

A - B / C / D - E - F.

It should be mentioned that a list can be recursive: it may contain a list (i. e. the name of the list enclosed in @ signs which, of course, refers to another list).

A type of list will be indicated, not by a series of combs which are not part of a keyboard; but by the particular list of the type where all of its components are empty; enclosed by two @ signs. For instance,

@ - / / - - @

designates a list schematized by:

2. Morphisms, List operators, and comb construction

2. 1 Morphisms or operations on lists

Let @ LIST 1 @, and @ LIST 2 @ be two lists. We will define the operation juxtaposition (putting them side by side) which we will designate by the sign / . For example:

@ LIST 1 @ / @ LIST 2 @ .

In a like manner, we will use the notation - to show the insertion of a word (or list) into another list. For instance the expression:

A - / B - C - @ X @ / @ Y @ UT

if we define:

@ X @ = D - E / F / G - H
@ Y @ = T and becomes
A - / B - C - D - E / F / G - H - TUT

2. 2 List ordering and comb scheme construction

Let us consider the following French sentence.

je ne veux pas .

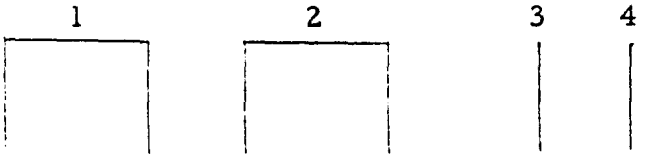
It is written in our alphabet as:

JE#NE#VEUX#PAS.

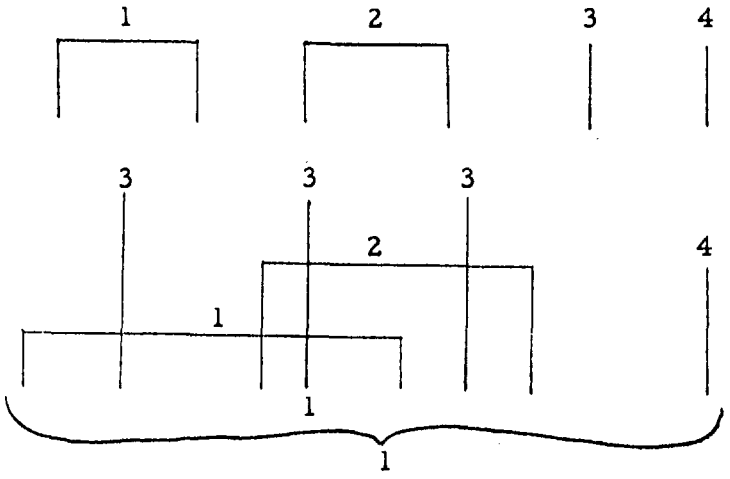
and this constitutes a word in our alphabet. It perhaps may be constructed with the aid of the following non-connected words:

(1) JE - VEUX
(2) NE - PAS
(3) # (non-connected word of one component)
(4) .

these words, thus ordered, constitutes a list schematized by :



Shuffled according to the following scheme: the non-connected words of the list, each one taken once or many times (the @ is taken three times), is shuffled into the components of one against the other: thus a juxtaposition of the various components in the order where the preceding shuffling places them.



The first line of the schematic represents the list from which we started. We will call it the source of the morphism.

The second line represents how we have shuffled the various teeth of the source. We will call it the graph of the morphism.

The third line represents the non-connected word (here of a single component). We will call it the target of the morphism.

The three line schematic represents the morphism.

It should be clear that since the first line of the schematic represents a list, the morphism can operate on any list of this type. Thus, the list:

A - B / C - D / E / F will become

A E C E B E D F .

The morphism conserves in each non-connected word the order of its components (JE preceeds VEUX, etc.). This is a condition we may or may not impose on a morphism. If we impose it, we are saying that it is forbidden to cross the teeth of the combs.

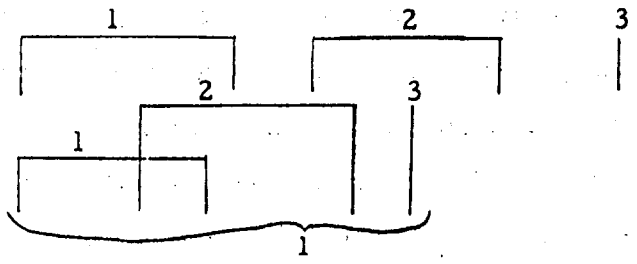
Let us note that we could also have constructed the sentence:

je ne veux pas .

by starting with the list:

JE # - VEUX / NE # - # PAS / .

which yields the following morphism:



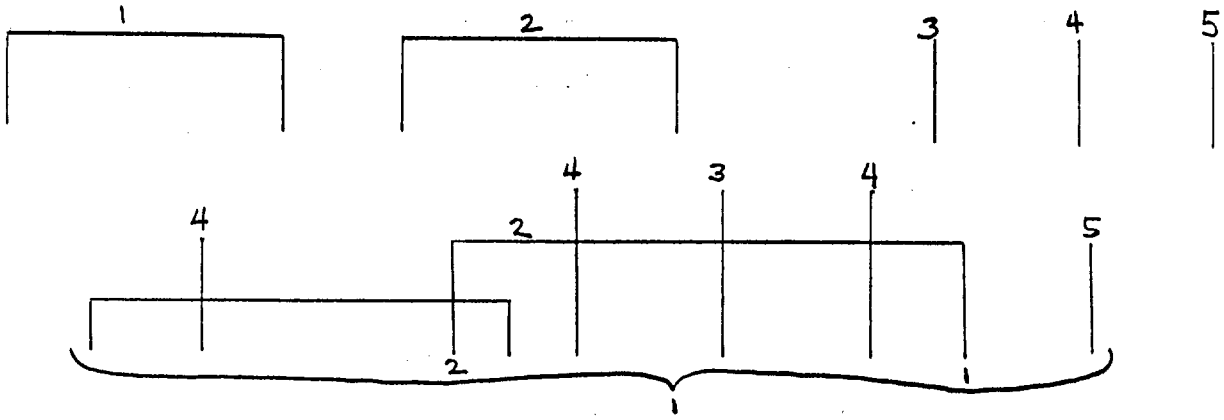
Let us briefly consider the following German sentence:

Du raümst dieses ein .

We could define it with the following list:

DU - ST / EIN - RAUM / DIESES / # / .

from which we obtain the following morphism.



Here the morphism commutes the teeth of the second comb of the source.

2. 3 Numerotizing the morphism

It is more convenient to use numbers instead of teeth and combs for the input to a computing machine. We have therefore designed the following type of numbering system for our morphisms.

We will define a morphism by a symbol (set of letters and digits). We shall precede this symbol by the sign $. For example:

$ PSI

will designate a morphism.

Let @ A @ be a list of the source type of $ PSI; the resultant list of @ A @ transformed by the morphism will have the following notation:

$ PSI (@ A @).

The morphism itself will be notated by the following type of its source list between two $ written as a series of integer triplets representing the series of teeth of the graph: the combs of the graph being ranked in the order of the encounter of their first teeth going from left to right. All of the teeth of the graph will be represented by three integers each one being separated by a comma:

1) the rank of the comb of the graph to which it belongs;
2) the rank of the corresponding comb of the graph;
3) the rank in the comb of the source of the tooth, corresponding to the given tooth of the graph.

The target and the correspondence between the graph will be indicated by the symbols

. - and /

placed between the triplets:

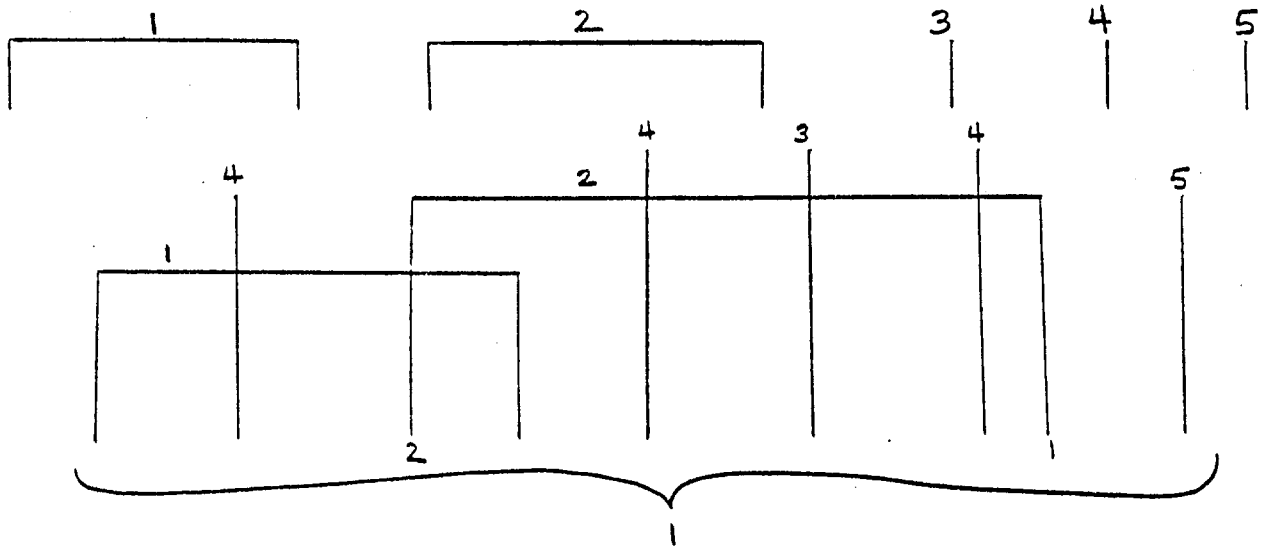
two triplets separated by the sign . correspond to a similar tooth of the target;

two triplets separated by the sign - correspond to two different teeth of the target belonging to a single comb;

two triplets separated by the sign / correspond to two different teeth belonging to two different combs;

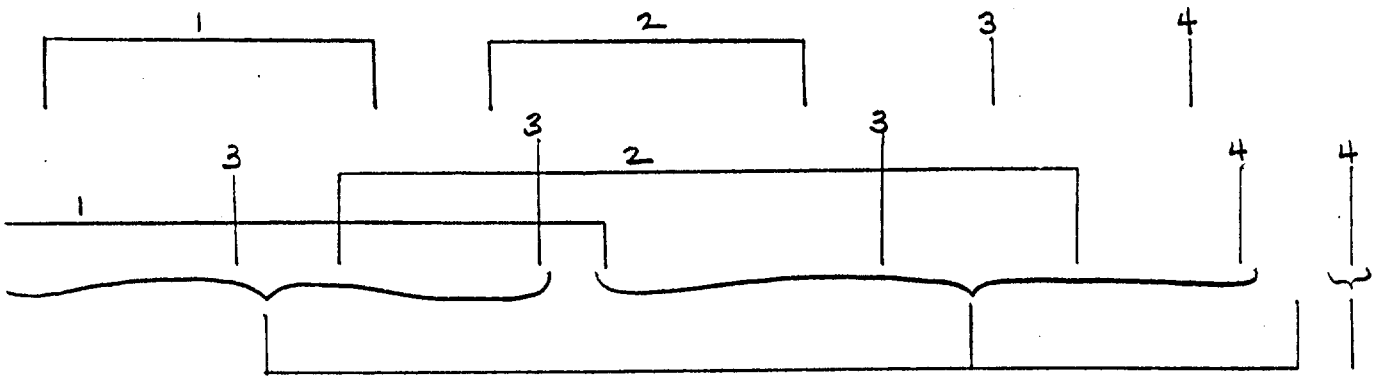
two triplets separated by the sign . will correspond to the same tooth of the target.

For example, let us describe the morphism for the previously mentioned German sentence: du raümst dieses ein. We shall call it $ PHI and schematize it as follows:



$PHI=@ - / - /// @ $ 1, 1, 1. 2, 4, 1. 3, 2, 2. 1, 1, 2. 4, 4, 1. 5, 3, 1. 6, 4, 1. 3, 2, 1. 7, 5, 1$

For a second example, let $ PSI be the morphism schematized by:



$ PSI=@ -/-//@ $1, 1, 3. 2, 3, 1. 3, 2, 1. 4, 3, 1-1, 1, 2. 5, 3, 1. 3, 2, 2. 6, 4, 1--/7, 4;1$

In this example, the third tooth of the first comb of the target does not correspond to any tooth of the graph; this is indicated in the series of integer triplets by the empty components having the sign / .

When we forbid the crossing of the teeth in the comb, the third number of the triplet representing a tooth of the graph is ambiguous for our notational purposes. We may, therefore, reduce our notation from triplets to couplets.

Thus $ PSI would become:

$ PSI = @ - /-// @ $ 1, 1. 2, 3. 3, 2. 4, 3-1, 1. 5, 3. 3, 2. 6, 4-/7, 4$.

2. 4 Operations on the lists by morphisms

In a precise manner, we can show that the scheme of a morphism $ RO is determined by the input of the lists. The first having the notation:

@ X @

composed of word-signs differing from each other and such that $ RO can operate on itself; the second, transformed from @ X @ by $ RO and having the notation:

@ Y @ = $ RO (@ X @) .

The source and the target of the scheme is none other than the schemes of @ X @ and @ Y @ respectively. This is due to the graph that is given by the comb-bindings of @ Y@ as each one can be identified, tooth by tooth, from a comb of the source.

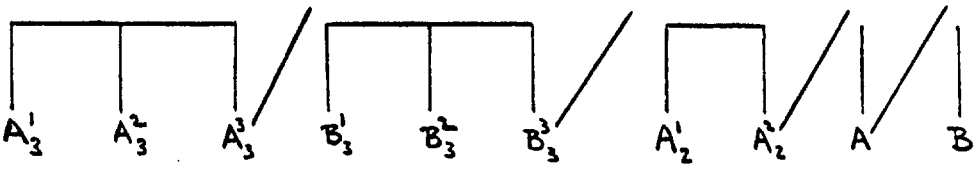
In effect, the comb schemes of @ X @ coincide with the comb-bindings; by series, if two letters of the substance @ Y @ of @ Y @ have the same path. The combs of a graph of $ RO are none other than the comb-bindings of @ Y @ in position. Moreover, since all the word signs of @ X @ are distinct between, we have the ability to distinguish the combs from each other. Thus, we know which comb of the source corresponds to which comb of the graph; and in this comb of the source which tooth corresponds

to which tooth under consideration; equally, we know, thanks to the - and / , the group of teeth of the graph which has joined the above bracketing of the teeth of the target.

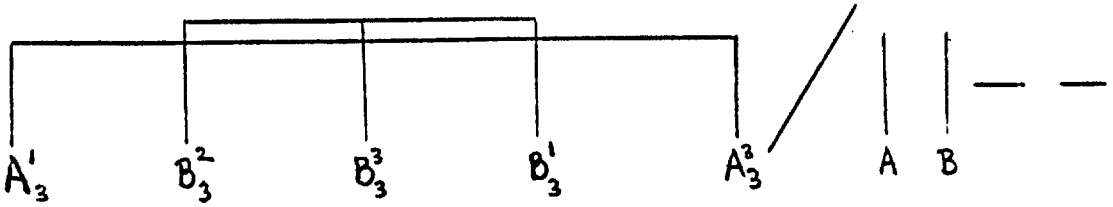
We can, therefore, reconstitute the complete scheme of the morphism $ RO.

Here is an example.

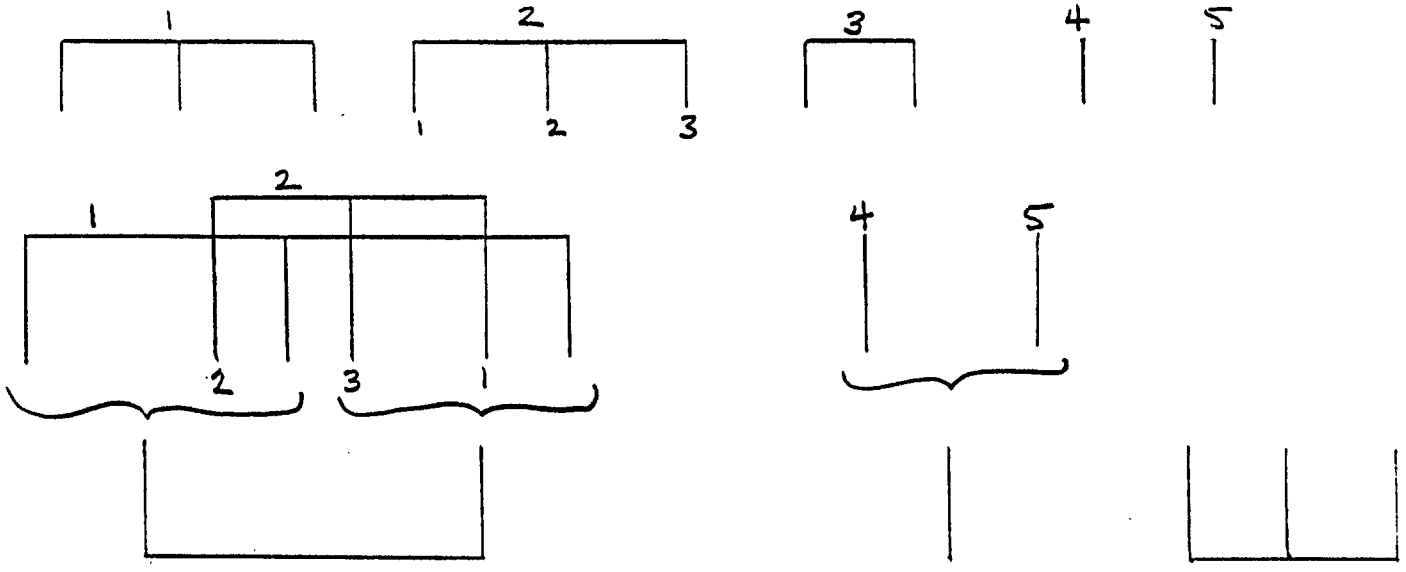
@ X @ =



@ Y @ =

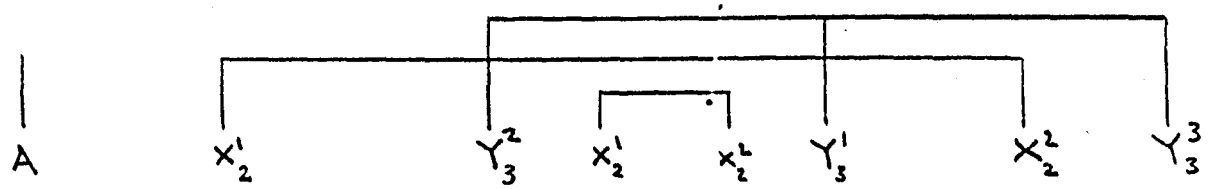


The morphism $ RO which transforms @ X @ into @ Y @ has for its scheme:

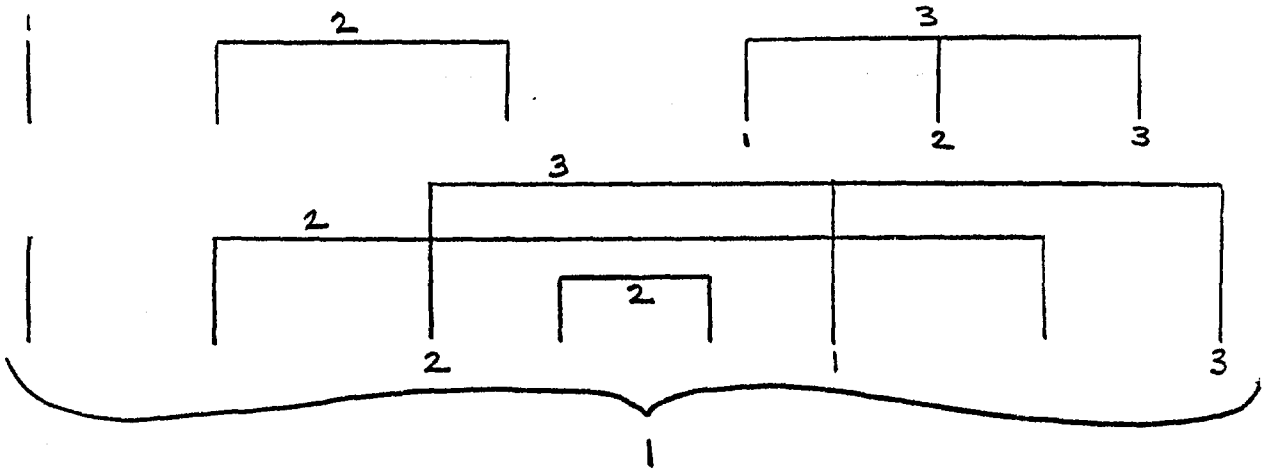


@ Y @ = $ RO (@ X @)

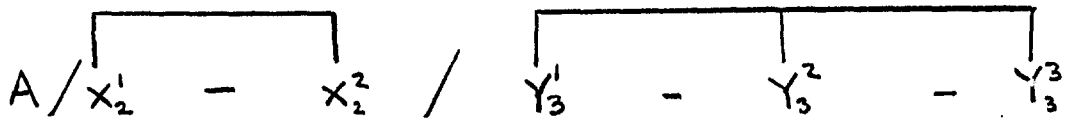
For a second example let @ M @ be:



and the word is transformed by the morphism $ RO of the following scheme:

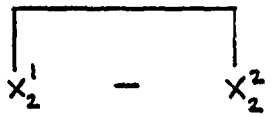


from the list:

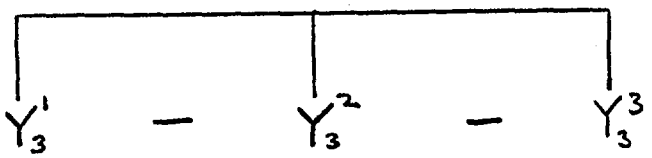


giving:

@ X @ =



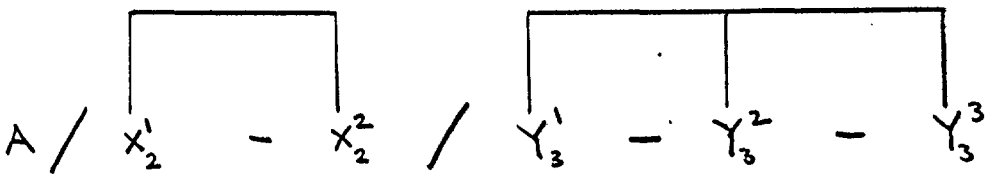
@ Y @ =



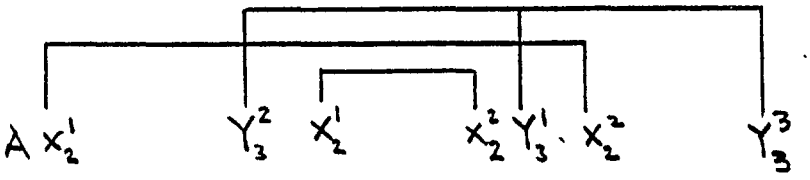
is the word @ M @ is none other than:

$ RO (A / @ X @ / @ Y @)

where A / @ X @ / @ Y @ designates the product of a juxtaposition:



The above is simply to say that there always exists a canonical form of this comb-scheme and list type structure. Our canonical form (going from left to right) for:



is as follows:

$ RO (A / @ X @ / @ Y @).

2.5 Operators defined from morphisms

We are going to define from the morphisms some operators on sets of lists. We will show later that the operators are themselves morphisms: therefore they are recursive and this then gives us the ability to have an extended logic for our computing language.

There are three operators for our morphisms: the / product, the // product and the * product (or composition).

I. The / product of two morphisms

Let $ RO and $ MU be two morphisms. The / product has the following notation:

($ RO / $ MU)

and is defined for any list @ X @ :

($RO/$MU)(@ X @) = $RO (@ X @) /$MU(@ X @)

(i. e. the list ransformed by $RO/$MU of @ X @ is the product of the juxtaposition of lists transformed from @ X @ by $ RO and $MU).

II. The // product of two morphisms

Let $RO and $KHI be two morphisms. We shall use the following notation.

($RO//$KHI).

For any lists @ X @ / @ Z @

($RO//$KHI)(@ X @ / @ Z @) = $RO (@ X @) / $KHI (@ Z @)

It should be mentioned that the // product of two morphisms is defined only when they are from the same source.

III. The * product (or composition) of two morphisms.

Let $RO and $PI be two morphisms such that the source of $RO is identical to the target of $PI. The * product has the following notation:

($RO * $PI)

For any list @ V @

($RO * $PI)(@ V @) = $RO($PI(@ V @))

It should be mentioned that since the * product of two morphisms is defined only if the source of $RO is identical to the target of $PI, it thus transforms any list of source type of $PI into a list of target type $RO.

2.6 Product of morphisms are themselves morphisms

The operators defined in 2.5 / product, // product, and * product of morphisms are morphisms.

In effect, they can be represented by schemes which are schemes of morphisms.

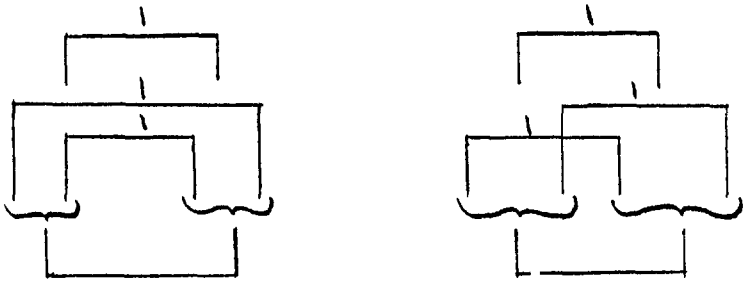
The scheme ($RO/$MU) is constructed as follows:

The first line is the source common to the schemes of $RO and $MU.

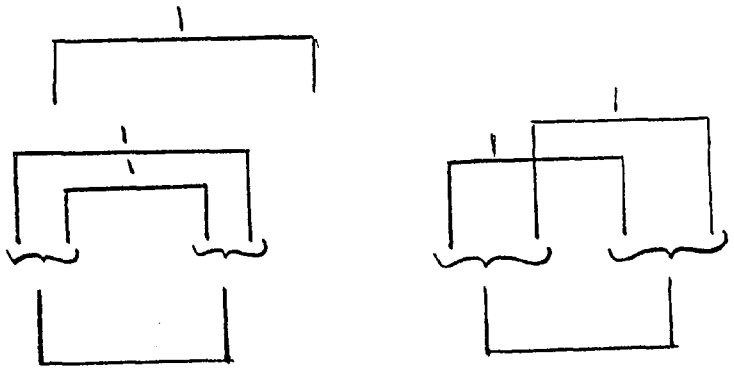
The second and are those of $RO and $MU placed side by side; those of $MU to the right of those of $RO.

For instance, given:

$RO: $MU:



The scheme of ($RO/$MU) is as follows:

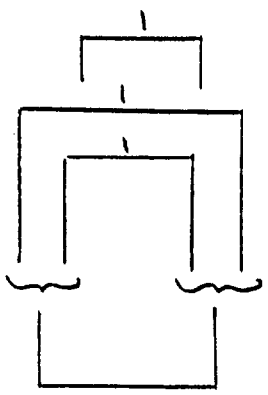


The schemes of ($RO//$KHI) are constructed as follows:

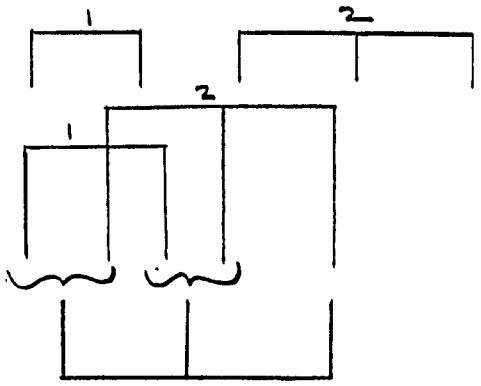
We modify the number of the combs of the scheme of $KHI while adding to each of them the number of combs of the source of $RO; thus, we can place the schemes of $RO and $KHI side by side; those of $KHI to the right to those of $RO.

For instance, given:

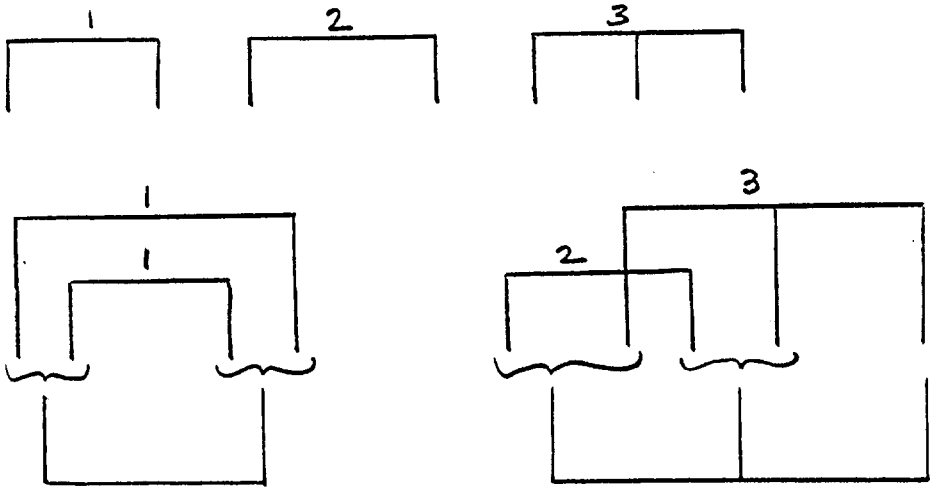
$RO:



$KHI:



The scheme for ($RO//$KHI) is as follows:



The scheme of ($RO * $PI) is constructed as follows:

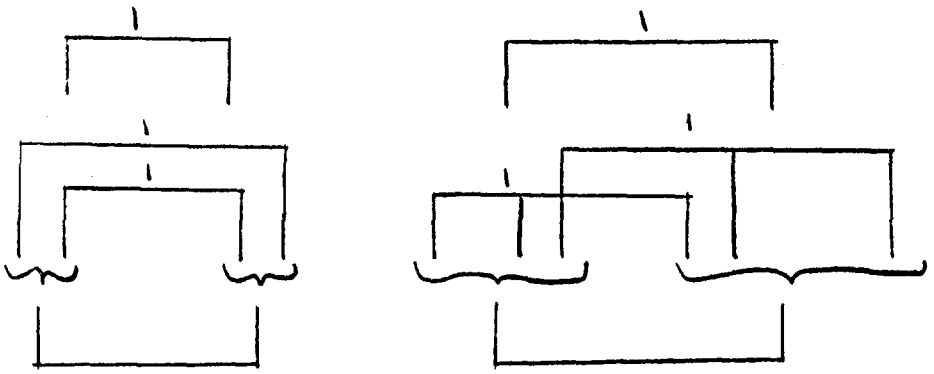
The source is that of $PI.

The graph is that of $RO where we replace each comb (which is also a comb of the target of $PI) by the system of combs to which it corresponds in the graph of $PI.

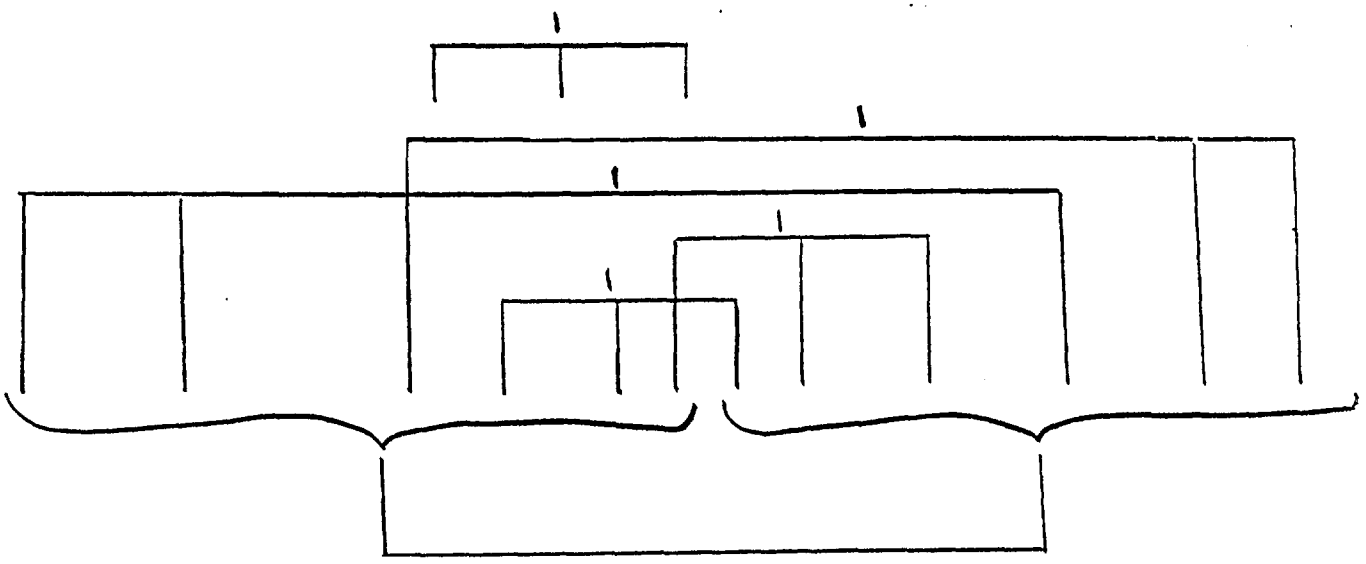
The target is that of $RO.

For instance, given:

$RO: $PI:



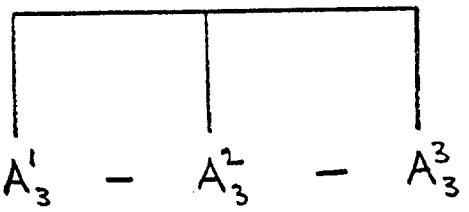
The scheme of ($RO * $PI) is as follows:



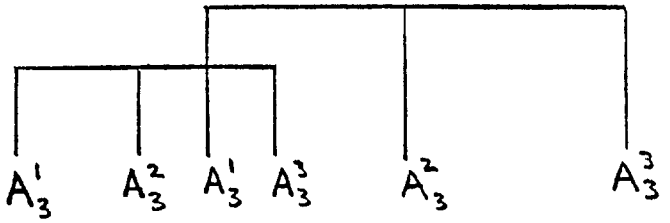
To construct the scheme of the * product of $RO and $PI, we can also characterize it by its action on a list @ X @ of source type of $PI: uniquely composed of word-signs different from each other. For that, we will make $PI act on @ X @, then $RO on $PI (@ X @).

Retaking the previous example; given:

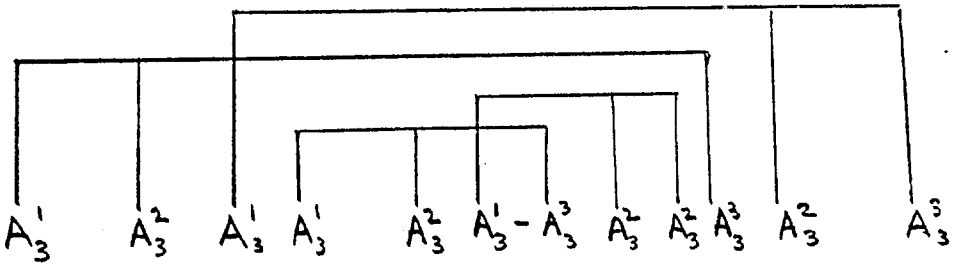
@ X @ =



$PI(@ X @) =



$RO($PI(@X@))=



from which is defined the scheme for ($RO * $PI).

The different products /, //, and * defined between the morphisms are not associative between themselves. They are associative when there intervenes only the product of a single sort (/, //, or *) which permits in this case the suppression of parenthesis. Thus:

(($RO * $PI) * $XI) = ($RO * ($PI * $KHI)) may also be written as:

$RO * $PI * XI .

3. Practical application of symbolic list processing

3. 1 Introduction

The actual programming of a symbolic list processing system presents many interesting and varied problems and possibilities. The power of this type of list manipulation opens the way to many hitherto unsolvable (computer-wise)problems.

Now that we have defined our language, it naturally behooves us to ask how, and to what, we can apply it. We are going to illustrate three different types of problems to show that we have a general purpose language. The first will be a simple example of list processing manipulating variables in algebra. The second will be an application of morphisms to generate FORTRAN (or any symbolic programming system) in the same general manner of SHADOW but with the added difference that our input is a list instead of a single variable. The third example will be an application to the natural languages.

3. 2 Operating on polynominials using symbolic list processing

We have written a program for the 1620 computer in FORTRAN which allows us to operate on sets of polynominials of N variables varied to a power p. The only limitation being that of memory size i. e. N^p size of memory. If we consider an external memory source such as tapes or discs this does not become much of a limitation.

The operations we are treating at present are addition, subtraction, multiplication and raising to a power. We are considering the differenciation but that we will leave to a later date. Mathematically expressed, we are doing the following:

Let $P_1(X_1, X_2, \ldots . X_N)$ and $P_2(X_1, X_2, \ldots . X_N)$

be polynominials when the variables can be raised to the j power and all combinations. An example will clarify this section. Let:

$$P_1 = X^3 + 5X^2 - 6X + 1 + Y^3 - CY + 4XY$$

$$P_2 = -3Y^2 - 6X^2Y + X$$

We are able to compute $P_1 * P_2$, $P_1 + P_2^2$, and any combination of these polynominials. It turns out that this can be handled very easily as a function of lists. Our limitations, at Rennes, being simply memory size as we need to set up tables for the coefficients and the 1620 is rather limited in memory capacity and lists are very heavy users of tables. This limitation would be relatively nil, if we had an external memory device such as discs or tapes.

3.3 Generating computes programs from flow charts.

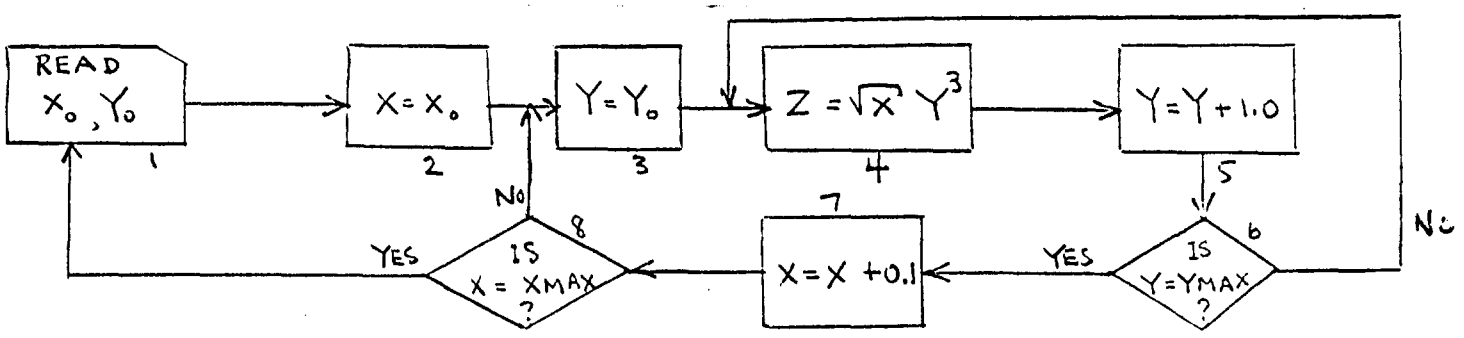
A practical application of the morphisms is the translation of flowcharts into programs. This section is therefore concerned with that problem.

A typical flowcharting or programming problem. Let us consider that we wish to solve the following general type problem:

$$Z = \sqrt{X}\ Y^3$$

$X = 0.(.1)20.0$ and $Y = -4.0(1.0)8.0$

the flowchart would appear to be of the following form(since there are no fixed rules for the logical flow or organization of a computer solvable problem, this flow chart is as good as any).

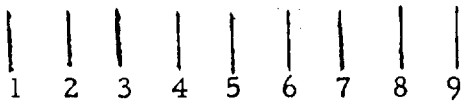


where: $X_0 = 0$, $X_{MAX} = 20.1$

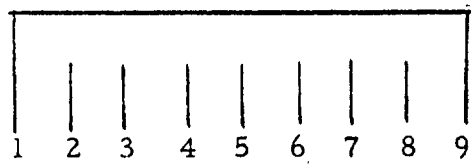
$Y_0 = -4$, $Y_{MAX} = 9$ for our original problem.

3.4 The comb-schematic

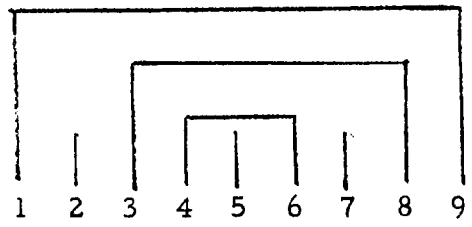
Let us number each box from left to right. Now let us draw 9 teeth and number then as follows (we need the nineth as there are two possible flows for the lost tooth).



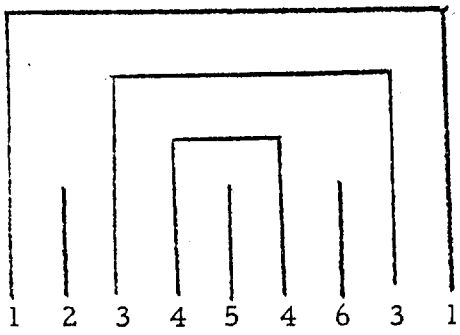
It follows that the 9th tooth meets the first, so let us join them:



Also, the 8th meets the 3rd, and the 4th meets the 6th. If we join them, we will have:



Now, for the sake of clarity, and to take advantage of our comb-scheme, we will re-number them as follows:



3.5 The list of our flowchart

For the list of our flowchart, it is necessary to write only the imperative instructions. We will generate the IF and the right paranthesis with the correct flow from our morphism. Therefore, the list is as follows:

```
READ 100, X0, XMAX, Y0, YMAX
X = X0
Y = Y0
Z = SQRTF (X) * Y * * 3
Y = Y + 1.0
Y - YMAX
X = X + 0.1
X - XMAX
O
```

the program will also generate an END statement.

The type of list is:

/////--//--//

Using the definition of a list as in section two, we have:

@ MU @ = READ 100, X0, XMAX, Y0, YMAX/X=x0/Y=Y0/Z=SQRTF(X) * Y * * 3/Y=Y+1.0/Y-YMAX/X=X+0.1/X-XMAX//$\bar{0}$

(the notation $\bar{0}$ is to signify the end of a list since a list may be of variable length. For descriptive purposes, we will continue to use the / to delimit an element of a list and a - (dash) to denote an insert. In practice, (for this particular problem), we cannot use these symbols as they are used by FORTRAN and we cannot distinguish whether they are for FORTRAN or for our list processing language.

3.6 The Morphism

As mentioned

$PI (@ MU @) = $1, 1.2, 1.3, 1.4, 1.5, 1.4, 2.6, 1.3, 2.1, 2.$ or

$PI (@ MU @) = @ /////--//--//@ = $1, 1.2, 1.3, 1.4, 1.5, 1.4, 2.6, 1.3, 2.1, 2$

(Since a morphism may be of any length, we must also define its length)

3.7 Decomposing the morphism

For convenience, let us re-order the morphism into tabular form.

i	B	C
1	1	1
2	2	1
3	3	1
4	4	1
5	5	1
6	4	2
7	6	1
8	3	2
9	1	1

We can scan the list C until we come to an entry 2. This will cause an IF(to be generated. We follow this by the entry in the list (in our case Y-YMAX). This is followed by a right paranthesis. The FORTRAN statement numbers are respectively the corresponding entry for B taken twice (separated by commas) and the value of i+1. This is true because all orders following an IF must have a path so they must be numbered i.e. carry a statement number.

3.8 Advantages and power of symbolic list processing with morphisms

With the example shown here, the real power does not come to light. However, there are some things we can do here. We can take advantage of inserts. We can insert elements into our list by adding an element to the end of the list and for putting a dash in the list; or we can even insert

a list /s within our original list by inserting the name of a new list /s enclosed in two @ signs.

It should also be mentioned here that it is not necessary to punch redundant coding in the above example. It suffices simply to add the number of the element to the morphism list. For instance in our example if we wanted to do another series of calculations over the same range, we need only punch a new entry for Z (or some other variable with another symbolic name) and repeat the tables.

With a little imagination, it is easy to see that if we can generate an IF statement, we can search to find out if our variable starts with the letters i, j, k, l, m, or n and generate a DO statement. Unfortunately, this, despite IBM's claim of FORTRAN's compatibility between machines, creates the problem of being machine oriented e.g. the 704/7090 etc. permit a DO loop to have a maximum value of 32,767 while the decimal machines will easily accept 99,999.

We could also attack the logical schematic of a flowchart with IF statements but this poses to the problem of ordinary usage and becomes completely computer oriented. For example, we can generate a statement of the following type:

(1) IF ( ) α , α , β

(2) IF ( ) α , β , α

(3) IF ( ) α , β , γ

(4) IF ( ) β , α, α

Which pattern shall we select? We can't accept the third one for a binary machine as floating point equality is a rather elusive animal due to binary round off.

This example was chosen to demonstrate that with list processing we can generate FORTRAN programs.

3.9 Linguistic Transformations

Let us state the following rules (not quite foreign to Ingwes):

@ SEN @ ⟶ $LA * $RO (@ SUB @ / @ VER @)

@ VER @ ⟶ @ VER_i @ (i=0, 1, 2, 3)

@ VER_j @ ⟶ $RO (@ MOD_j @ / @ VER_i @)

@ SUB @ ⟶ # THEY-
# HE-S
# THE-LINGUISTS-

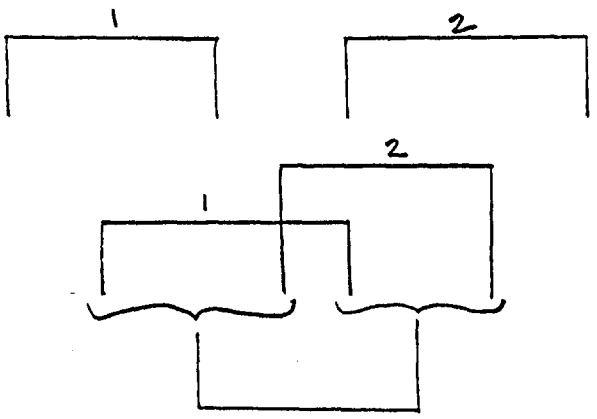
@ VER0 @ ⟶ # INSERT - # (or # CLIMB-UP #...)

@ MOD1 @ ⟶ # BE-EN

@ MOD2 @ ⟶ # BE-ING

@ MOD3 @ ⟶ # HAVE-EN

$RO is a schuffling which out of two intercepted words makes a third one, according to the following scheme:



: e.g. $RO(A-B-/C-D) = A-CBD

$LA is a "junction" of the two components in one: e.g. $LA(A-B) = AB.

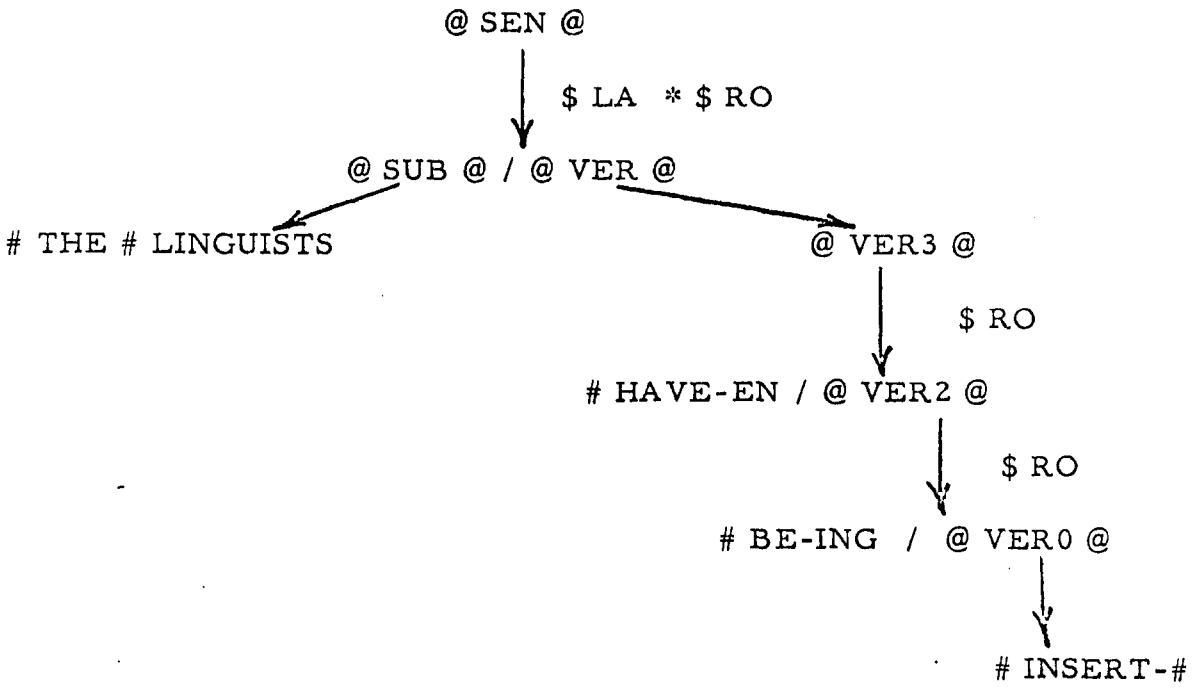
Now we can derive a sentence (morpho-phonemic rules are omitted in our sketch, they happen to be useless for this example...).

@ SEN @ ⟶ $LA * $RO (# THE # LINGUISTS / $RO
(@ MOD3 @ / $RO (@ MOD2 @ / # INSERT)))
@ SEN @ ⟶ $LA * $RO (@ SUB @ / $RO (@ MOD3 @ /
$RO (@ MOD2 @ / @ VER0 @)))

that is e. g.

THE # LINGUISTS # HAVE # BEEN # INSERTING

or in the shape of a tree.



Bibliography

J.P. Benzécri, 3° leçon (course and publication de la Faculté des Sciences de l'Universite de Rennes, France).

F. Benzécri; Linguistique Mathématique (publication de la Faculté des Sciences de l'Université de Rennes, France).

See also paper being presented at the 1965 International Federation of Information Societes Congress (IFIPS) in New York by Professor J.P. Benzécri particularment for linguistical applications.