[*International Conference on Machine Translation of Languages and Applied Language Analysis*, National Physical Laboratory, Teddington, UK, 5-8 September 1961]

THE MECHANICAL ANALYSIS OF LANGUAGE

by

MICHAEL LEVISON, B.Sc.

(Department of Numerical Automation,
Birkbeck College, University of London)

INTRODUCTION

AFTER the early heuristic attempts at crude mechanical translation, it was clear that the next step was to insert into the machine programmes containing detailed grammatical and syntactic information regarding the particular languages being processed. Unfortunately it was found that this information was not available in a sufficiently logical form to be taken over into machine programmes in a relatively simple way.

To overcome this difficulty work has been carried out in the Department of Numerical Automation, Birkbeck College, to develop programmes which will themselves make analysis of language in terms directly suitable for use by a computer.

NOTATION

The following notation; previously adopted by the author,[1] will be used herein:

(1) $W_\alpha << W_\beta$, to denote that word W_α is alphabetically earlier than word W .

$W_\alpha >> W_\beta$ and $W_\alpha = W_\beta$ are also used appropriately.

(ii) *Flow Diagram Conventions:*

E is the point of entry into the programme. Other circles contain alternative points of exit, or points referred to in the text of the paper.

Blocks with a double line to the left contain the initial settings of the various parameters.

Diamonds contain questions with Yes/No answers.

ALPHABETIC ARRANGEMENT

The simplest of all applications of a computer to linguistic analysis are such problems as the evaluation of the relative frequencies of various letters, letter-groups or initial letters in a given text, and the calculation of the average number of letters per word or words per sentence. These problems are so simple that no comment is needed as to their solution, though it should be noted that the results, especially of the latter two, are of interest in analysis of literary style.

A problem of somewhat more complexity is that of alphabetic arrangement. It is often useful or necessary to be able to arrange in alphabetical order a set of words in the store of the computer.

One possible programme for this purpose is illustrated in *figure 1.* The number of words in the set is n, and these are considered initially to occupy "dictionary positions" in the store numbered from 0 to (n-1). D_p denotes the contents of dictionary position p at any stage of the programme.

The process involved is to find, for each r from 0 to (n-2) in turn, the alphabetically earliest of the words (D_r, D_{r+1}, ..., Dn-1), and to interchange this with D_r.

At stage A, s indicates the position of the alphabetically earliest word so far found for this value of r, and t the position of the word with which it will be next compared.

The programme requires

$$\sum_{r=o}^{n-2} \left\{ (n-l)-(r+l) +l \right\}$$

$$= \tfrac{1}{2}\,(n-l)n$$

word comparisons, and (n-1) word interchanges.

Other programmes for the same purpose can be based on glossary-making procedures.

If a further n dictionary positions are available, a glossary of words can be constructed in these extra positions. A typical programme, involving for each word in succession logarithmic searching of the extra positions followed by shifting and insertion, would require, on average, approximately n $\log_2$ n word comparisons and $\frac{1}{4}n^2$ word shifts.

If the extra dictionary positions are not available, a possible programme would be to insert each successive word starting with the second in its correct alphabetical position among the previous words. This would require, on average, approximately $\frac{1}{4}n^2$ word comparisons and $\frac{1}{4}n^2$ word shifts.

Choice of programme is determined by available space and by time, the latter being governed largely by the times of the word comparison and word shift operations. In this connection it should be noted that one interchange is usually equivalent to three shifts, that is

$$(A \leftrightarrow B) = (A \rightarrow x) + (B \rightarrow A) + (x \rightarrow B)$$

In practice word comparison is almost always a somewhat longer operation than word shifting, so that, if the necessary space is available, the second of these three processes is usually the fastest.

GLOSSARIES AND CONCORDANCES

The construction of a glossary of a given text - that is, an alphabetical list of the different words occurring in the text, together (usually) with a count of the number of occurrences of each word - is often performed manually using a card index. The text words are examined consecutively and each new word is written on a new card which is inserted in its correct alphabetical position in the index.

Where punched card equipment is available, a more mechanised procedure can be used, the text words being punched each on a separate card and the cards being sorted into alphabetical order using a punched card sorter.

Both of these systems can be adapted for use on a computer.

In the latter case the words of the text can be read to the computer store and then sorted using one of the procedures described in the previous section. For reasons derived from the final paragraph of that section however, this programme would usually be slower than the "card index" programme described below.

In the "card index" system, the different text words so far found are stored in alphabetical order in consecutive positions of the computer store, each accompanied by a word count showing the number of occurrences to date. Any incoming word is looked up in this list. If no match is found, it is inserted in its appropriate position in the store, the words below it being shifted down. If a match is found, 1 is added to the word count of the matching word.

The dictionary search is best conducted using the logarithmic search described in some detail in a previous paper. [1)] A small modification must be made to that programme to enable it to indicate the appropriate dictionary position *for* words not matched.

In adapting the "card index" system for a computer two difficulties are encountered, namely

(1) that the (medium access) storage of a computer is, unlike a card index, not unlimited in size; and

(ii) that the necessity of shifting all words alphabetically later makes the insertion of a word into the dictionary a relatively long operation.

A number of methods are available to assist in overcoming these problems; and some of these have been described by A.J.T. Colin [2)] and others.

To overcome the problem of limited store size the glossary must be constructed in several "cycles", each cycle giving rise to a "sub-glossary". These must then be combined to form the complete glossary.

Two methods of achieving this are for the programmer (1) to pre-divide the text into sections and produce a full-alphabet glossary for each, or (2) to pre-divide the alphabetic range into sections and produce a full-text glossary for each.

Methods not involving this pre-division follow the lines described above until the store is full, at which stage several alternatives are possible.

In alternative (3) the first cycle ends here and text input ceases while the contents of the store are output as the first sub-glossary. The store is then cleared and text input resumes for the next cycle.

In alternative (4) text input continues, any further unmatched word being punched on an "overflow" tape, which forms the text for the next cycle.

In alternative (5) also the text input continues, but any further unmatched word is inserted in its correct position, the final word of the store (possibly the input word itself) being ejected. For the next cycle the whole text is re-read, but words in the alphabetic range spanned by the previous sub-glossary are ignored.

A further variation of (2) or (5) is to punch those words omitted or ejected from the store during the present cycle (together, if necessary, with their word counts to date) on an overflow tape, this replacing the text for the next cycle.

In (1), (3), (4) the sub-glossaries must be "meshed" in order to produce the complete glossary. The meshing can, of course, be performed on the computer.

(1), (3) require more cycles than the other alternatives, since, in these cases, many words will occur in several cycles. These alternatives have the advantage, however, that each text word need be input once only.

The times taken by each of the alternatives could, of course, be compared theoretically, but the number of assumptions necessary would make the conclusions of doubtful validity.

Let us now consider the problem of word shifting. The average number of words which must be shifted in order to insert the kth new word into the store is $\frac{1}{2}k$. Thus, to fill a store of capacity n words would require on average

$$\sum_{k=1}^{n} \tfrac{1}{2}k \cong \tfrac{1}{4}n^2$$

word shifts.

Suppose, however, that the store is divided into r equal substores, and the (current) alphabetical range into r approximately equal Intervals, each Interval being collected in a separate substore.

Then the filling of each substore requires on average $\frac{1}{4}(n/r)^2$ word shifts, and the filling of the whole store $\frac{1}{4}\, n^2/r$.

This device to reduce shifting can be combined with any of the alternatives described for coping with the problem of limited storage.

The Logical Tree Process

A method of constructing a glossary on a computer basically different to those described above is the "logical tree" process,3) which enables word shifting to be eliminated completely.

In this method each different word is stored not only with a word count but also with two spaces for address references which are Initially blank. The first word, Do*,* of the text is read and stored in the first dictionary position.

Each subsequent text word, W_T, is read and compared initially with D_O.

(i) If W_T - D_O, 1 is added to the word count of D_O and the programme returns to read the next text word.

(ii) If $W_T << D_O$, then the first of the two address references of D_O is examined. If this is found to be blank, then W_T is inserted in the next available dictionary position of the store. The address of this position is then inserted as the first address reference of D_O, and the programme returns to read the next text word. If, however, the first reference of D_O is found to contain an address, then W_T is next compared with the word D_p indicated by this address, etc.

(iii) If $W_T >> D_O$, the programme follows a path similar to (ii), but using the second address reference of D_o instead of the first.

The process stops when the whole text has been read, or when the store is full. The latter situation may be handled by utilising any of alternatives (1), (2), (3), or (4) of the card index process.

After any cycle, the different words forming the "tree" can be retrieved in alphabetical order and output, without further word comparison, by applying a set of rules to the address references.

In order to consider the problem of alphabetical retrieval, it is convenient to represent the tree by a pyramid diagram.

Each word of the tree is denoted by a dot, the apex dot denoting D_O. The words W_1, W_2 indicated by the first and second address references of a word W are linked to W respectively by left- and right-inclined arrows, the arrows being directed away from W. *(figure 2a).*

Then, by the principle of construction of the tree, all words comprising the sub-tree leading from W via the left-inclined arrow are alphabetically before W, while those of the sub-tree leading from W via the right-inclined arrow are alphabetically after W. A typical tree is shown in *figure 2b,* the numbers indicating the alphabetical order of the words.

One set of rules for obtaining the alphabetical order of the tree is then as follows:

Suppose the m^{th} level to be the lowest level of the tree which contains a word. (In *figure 2b,* m = 5). If every dot on levels 1, 2, ..., (m-1) of the tree has both a left- and a right-inclined arrow leading away from it, the tree is said to be "complete" If not, the tree is made complete by the addition of further dots not representing words, these dots being linked to the tree by *broken* arrows.

Level s now contains 2^{s-1} dots, and these are numbered from left to right starting at 2^{m-s} and increasing by intervals 2^{m-s+1} to $(2^m - 2^{m-s})$, where s = 1, 2, ..., m. *(figure 2c).*

Then the alphabetical order of the original tree is the numerical order of the dots of the expanded tree with the added dots omitted.

In order to see how these rules may be used on the computer, it is necessary to note the following points:

(1) The (p + l)th dot from the left on level s of the completed tree is numbered

$$2^{m-s} + p.2^{m-s+1} = (2p+l)\ 2^{m-s}$$

Conversely, if m is given and n is any integer $< 2^m$, it is very simple to find the unique pair of integers (s, p) such that

$$n = (2p+ l)2^{m-s}$$

(11) Any dot of the tree can be specified by a sequence of the letters L and R (its "vector") indicating the directions of successive diagonals by which it is reached starting from the apex. The vectors of dots on level s contain (s-1) letters, and the (p+1)th dot from the left on this level of the completed tree has the vector obtained by writing p as an (s-1) digit binary integer with L, R in place of 0, 1 respectively.

Thus, in *figure 2c,* for n = 14 we have s = 4, p = 3 and the corresponding vector is LRR.

(iii) The dot of the original tree corresponding to any given dot of the completed tree is reached merely by following the same vector through the original tree. The added dots of the completed tree are those whose vectors indicate in the original tree an arrow which is not present.

Now it is clear that the value of m may be ascertained during the compilation of the tree.

Thus alphabetical retrieval will be achieved if, for each n in succession from 1 to 2^m - 1, we evaluate (s, p), produce the corresponding vector, and follow this through the original tree.

An alternative process for alphabetical retrieval, not requiring the concept of the completed tree, also makes use of the pyramid diagram. The process involves a tour of the tree according to the rules given below.

The computer "travels" by keeping in its store a record of the path which must be taken to reach its current position from the apex. The path, in this case, is recorded as a sequence of addresses of the words through which it passes, any two being separated by a letter L or R indicating the direction of the arrow joining them.

The current position of the computer is the current last address of the path. For the computer to travel down the tree, addresses are added to the path; for it to travel up the tree, addresses are deleted.

The rules (which are obeyed in sequence unless otherwise stated) are as follows:

Rule 1 Start at the apex.

Rule 2 Travel leftwards down the tree stopping when a word is found with no left arrow leaving it.

Rule 3 Print this word.

Rule 4a If this word has a right arrow leaving it, travel along this arrow. Then jump to *rule 2.*

b If not, return up the tree until *one* left arrow has been ascended. Then jump to *rule 3.*

c If the apex is reached without a left arrow being ascended, then all the words have been printed and the process stops.

Logarithmic Construction Process

The glossary construction procedure described above was designed specifically for use on a computing machine. Another such process which also eliminates all word shifting, makes use of the logarithmic dictionary searching procedure in a manner very different from that described earlier.

The process takes its simplest form if we allot for the storage of glossary words N storage positions, where

$$N = 2^m - 1 \text{ (m integral)}$$

This choice of N is always possible, but may lead to a wastage of up to half the available store space. In this case some modification can be made to the process; for example, preparing a glossary of words with initial letters A-P in a block of $(2^m - 1)$ positions, and a glossary of words with initial letters Q - Z simultaneously in another block of $(2^n - 1)$ positions.

Suppose, therefore, that $N = 2^m - 1$, and that the N positions are numbered $0, 1 \ldots 2^m - 2$.

Then the first word of text is read, and is inserted in position $P_1 = 2^{m-1} - 1$ (the half~way position), together with a word count which is initially l.

Each subsequent word, W_T, of text is read, and compared, in the first place, with the word G_1 in position P_1.

(1) If $W_T = G_1$, 1 is added to the word count of G_1 and the programme returns to read the next text word.

(11) If $W_T \neq G_1$, the contents of position P_2 are examined, where
$P_2 = P_l - 2^{m-2}$ (the quarter-way position) if $W_T << G_1$
$= P_1 + 2^{m-2}$ (the three-quarter position) if $W_T >> G_1$
If P_2 is vacant, W_T is inserted into position P_2, and the programme returns to read the next text word.
If P_2 contains a word G_2, then W_T is next compared with G_2 and the programme continues as before, but with G_2 instead of G_1 and P_3 instead of P_2 (where $P_3 = P_2 \pm 2^{m-3}$).

(iii) If no vacant position has been found after successive examination of P_l P_2, ... P_m, then the word W_T is punched on an overflow tape, and the programme continues with the next word of text.

The process stops when the whole of the text has been read. A glossary of the overflow tape must then be made, and this is meshed with the initial glossary (and with any further overflow glossaries) to produce the final glossary of the text.

Successive positions P_l P_2, ... P_m are given by

$P_1 = 2^{m-1} - 1$, $P_s = P_{S-1} \pm 2^{m-s}$ (s = 2, 3, ... m)

the sign being positive if $W_T >> G_{s-1}$ and negative if $W_T << G_{S-1}$ where G_{S-1} is the word in position P_{s-1}. The 2^{s-1} positions P_s (that is, positions

$2^{m-1} \pm 2^{m-2} \pm 2^{m-s} - 1$ are said to be on the sth level.

The advantages of this process over the logical tree procedure are

(1) that the words contained in successive positions 0, 1, 2, ... (2^m - 2) are in alphabetical order, so that no complex alphabetical retrieval programme is required, and

(2) that no address references are stored, so that each store position takes less store space, and more positions can be fitted into the space available.

A disadvantage of the process is, however, the possibility of inefficient filling of the store. For example, if the first text word encountered is "YARD", most of positions 2^{m-1} to (2^m - 2) will remain unfilled throughout the programme, as will most of positions 2^{m-2} to (2^{m-1} - 2), if the first two words are "MAST" and "MASS" respectively.

Actually the first example illustrates about the worst possible occurrence and the effect of any such occurrences can be minimised by giving the glossary an "artificial start" of the type described for the tree process.[3] The effect will also be less serious if the number of different text words is much larger than (2^m - 1) for, in such a case, even the number of different words beginning with Y or Z may become large enough to fill much of the "semi-isolated" part of the store.

A further precaution which can be included in the programme is the rejection of W_T to the overflow tape instead of its insertion into a vacant position on the s^th level if its "alphabetical distance" from the word with which it was compared on level (s - l) is less than $\delta(s)$ or greater than Δ (s), where δ and Δ are some suitably chosen functions of s.

In this connection, a precise definition of the "alphabetical distance" of two words W_α and W_β, convenient if the words are represented numerically in the manner of the earlier paper,[1] might perhaps be

$$\left| \left(\sum_{r=0}^{j-1} 2^{-5r} \alpha_r \right) - \left(\sum_{r=0}^{k-1} 2^{-5r} \beta_r \right) \right|$$

where $\alpha_0, \alpha_1, \ . \ . \ . \ , \alpha_{j-1}$ and $\beta_0, \beta_1, \ . \ . \ . \ \beta_{k-1}$ are the numbers representing successive letters of W_α and W_β respectively. (The expression is, in effect, the difference between the numbers $\alpha_0, \alpha_1, \alpha_2 \ldots, \alpha_{j-1}$ and $\beta_0, \beta_1, \beta_2, \ . \ . \ . \ \beta_{k-1}$ in the scale of 32.)

In choosing δ and Δ it should be noted that there is nothing to be gained by rejecting a word from insertion into level m. It should also be remembered, when meshing the initial and overflow glossaries, that if a word is rejected at one level some later occurrence of it may be accepted for a deeper level. Thus it is possible for some occurrences of a word to be in the initial glossary and some in the overflow.

Much time will be saved in both of the preceding construction processes if the word "A" is given separate treatment.

The manual construction of a concordance of a text - that it, a glossary together with a page/line reference for each occurrence of every word - is similar to that of a glossary. However, the problems of shifting and overflow experienced in the automatic construction are far more acute than the corresponding problems in glossary construction.

In this case the most convenient method of overcoming the difficulties has been found to be the prior construction of a glossary.[2)]

A variation on the usual type of concordance found useful in the grammatical and stylistic analysis of a text contains, not the page/line reference of the various words, but their position numbers counting from the beginning of a text.

Such a concordance is constructed in a precisely similar manner to the more orthodox type, but contains slightly more information. It is useful, for example, for determining how many times a given word, or class or words, occurs in a given block of 1,000 words. It can be made to contain all the information of an orthodox concordance if during construction there is tabulated the page/line reference of every 200th word or the position number of the first word in every tenth line.

CONCLUSION

A number of tests are in the course of preparation to enable comparisons to be made of the relative efficiency of the various glossary procedures described above. It is hoped to publish the results of this investigation in due course.

REFERENCES

1. LEVISON M., *Information and Control,* 1960, **3**, 231-247.

2. COLIN, A.J.T., *Information and Control,* 1960, **3**, 211-230

3. BOOTH. A.D., and COLIN, A.J.T., *Information and Control,* 1960, **3**, 327-334.

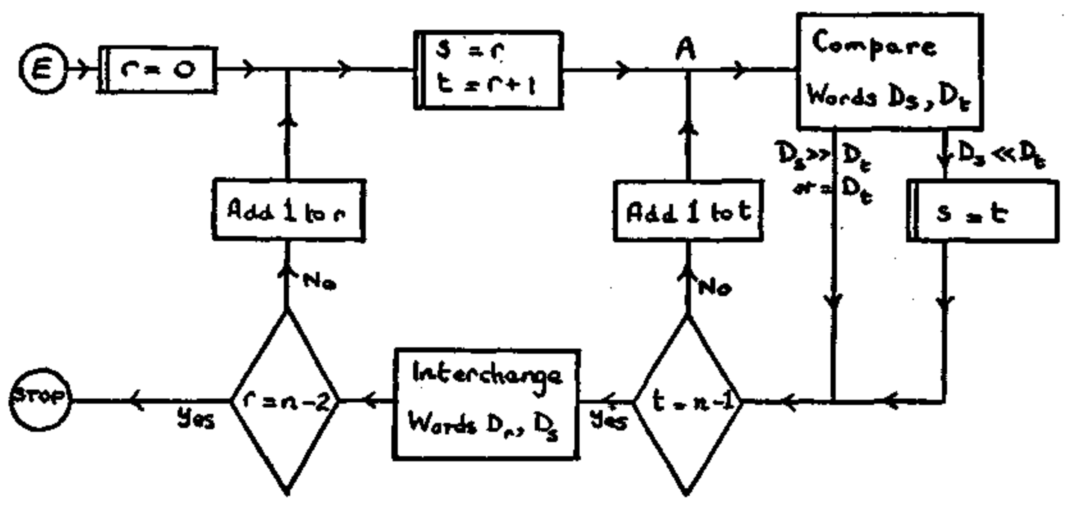

Figure l.

Alphabetic arrangement

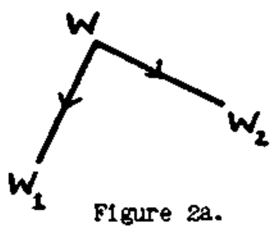

Figure 2a.

An Illustration of the tree diagram convention.

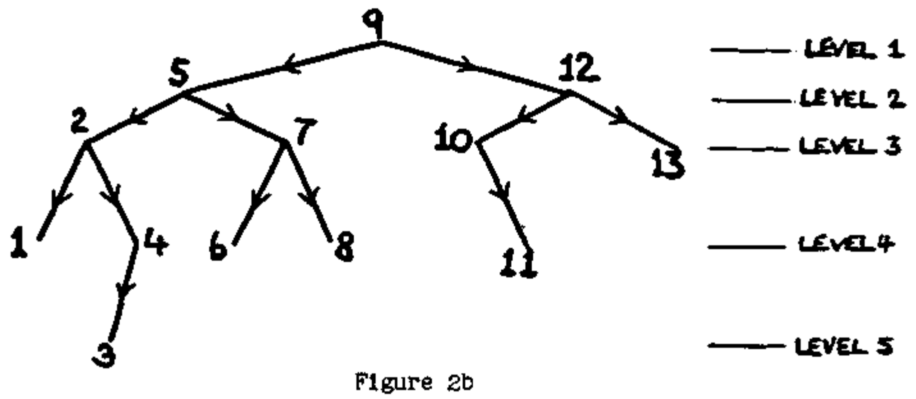

Figure 2b

A typical tree.

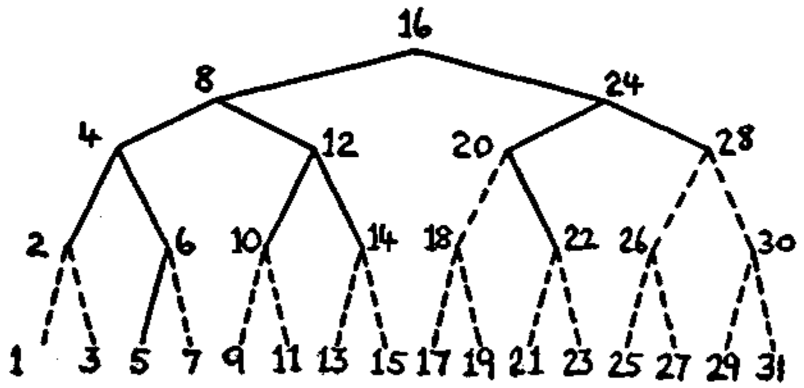

Figure 2c

The tree of Figure 2b, complete and numbered (with arrow-heads omitted).