[*Information processing: proceedings of the International Conference on Information Processing*, Unesco, Paris 1959] 163

The COMIT system for mechanical translation

By V.H. Yngve,

Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts (USA)

The new MIT programming language for mechanical translation is described and discussed. This language is being made the basis of an automatic programming system. The programming of the compiler-interpreter by the MIT Computation Center Staff is well underway and may be completed by the time of the meeting. The programming language is quite different from other programming languages because of its different purpose. The main features and advantages of the language are discussed in some detail together with a discussion of the considerations underlying the choice of these particular features, and examples of their use in programming linguistic problems.

A number of linguists have already been introduced to the programming language; a complete programmer's manual is available. The language is being used extensively in anticipation of the completion of the compiler-interpreter. How the language is working out in actual use is discussed.

1. Introduction

The field of mechanical translation (MT) has advanced to the point where a number of groups are programming experimental translating systems on general-purpose computers. Our imperfect understanding of the problem leads to the inevitable result that these programs are inadequate in many respects, our steady advance in understanding leads to our desire to write new experimental translating routines to replace the old ones while the extreme complexity of human language leads to large complicated programs that are time-consuming to write. In view of these considerations, the Mechanical Translation Group at MIT[1]) has undertaken to design a programming language [2] tailored to the needs of the problem, a programming language that gives the linguist direct access to the computer without his having to concern himself with details that are irrelevant to his problem. The language is being provided with a compiler and interpretive routine written for the IBM 704 computer by the Programming Research Staff of the MIT Computation Center.

2. The need

The reason that the mechanical translation programs being written today are of a trial or tentative nature is that human language and the process of translation are not well understood. Progress in research requires the tentative compilation of dictionaries and the tentative compilation of rules. The rules are not mutually independent but make up an intricate network of interdependencies, while frequent tests are necessary to establish the validity or lack of validity of the compilations. The lack of independence of the rules makes it unrealistic to insist that a translating program have the property that one can add to it by simple accretion when new facts about translation are discovered. For this reason, advances in our knowledge will usually require a complete reprogramming of a translation routine from the beginning.

We are thus faced with a programming effort of considerable magnitude, one in which the economies of an automatic programming system would be particularly valuable.Without such a system, each trial would yield small results for a large amount of effort. There is a further advantage in a system in which the linguist can easily do his own programming. In the past, linguists and programmers trying to work together in teams have suffered from a difficulty more basic than the usual difficulties of communication between experts in separate fields: Neither the linguist nor the programmer has been able to be fully effective. The linguist, not having an intimate knowledge of the capabilities of the machine, was unable to avail himself of its full power. The programmer, not having an expert knowledge of linguistic matters, was not easily able to use his special knowledge of the machine for the solution of linguistic problems. An automatic programming system which gives the linguist direct access to his machine by automatically taking care of the numerous details that are not an essential part of the problem, should greatly facilitate research in mechanical translation.

3. Specifications

We are thus led to set up the following requirements for an automatic programming system. These fall under the three, headings of utility, convenience, and simplicity.

1) The full utility or general-purpose nature of the computer must be maintained. We do not yet know exactly what linguistic operations will be necessary for effective mechanical translation but since the system is to be used for research purposes, it must be possible to express any operations that may be needed in the future. The general-purpose nature of the computer, then, must not be sacrificed when we design a system that meets the additional requirements of convenience and simplicity.
2) To meet the requirement of convenience, we must have a system that the linguist can use himself, a system adapted to his special purposes. We therefore have to foresee the kinds of operations that the linguist will want to carry out most frequently and make them easy to write. We want to provide special facilities as far as possible without destroying the general-purpose nature of the system and without encroaching too much on the simplicity of the system. The speed of operation of the final programs and the efficient use of computer storage, though certainly not to be neglected, are to be given secondary consideration to the convenience of the programmer; we desire a user-oriented system rather than a machine-oriented system. As an example of the sort of thing that is important, we would like a number of aids in checking programs, such as built-in automatic checks on the meaningfulness of the instructions, and a convenient method for printing out intermediate results. We want to relieve the programmer of the burdensome details of a computer-oriented system. We want him to be the easy master of the machine, free to exercise his creative ability.

[1]) The author is particularly indebted to G. H. Matthews for his many important contributions and to S. F. Best, F. C. Helwig, A. Siegel, and M. R. Weinstein of the MIT Computation Center for their many helpful criticisms and suggestions. Some of the features of the notation used by N. Chomsky in his theory of grammar have been incorporated. See [1].

3) Simplicity is desired so that the system can be easily learned. Extreme simplicity can be had only at the expense of convenience because it implies a few elementary operations rather than many convenient special-purpose operations. But simplicity of this sort is a false simplicity from the point of view of the programmer because he has to learn how to combine the elementary operations in special ways for his special purposes. A number of carefully chosen special-purpose operations can therefore actually add to the over-all simplicity of the system from the user's point of view. If we can design a system that utilizes the prospective programmer's knowledge of natural languages and his habitual means of expression in his field of specialization, we can achieve a system that seems simple and easy to learn in spite of its being convenient and therefore complex.

4. The system

In line with the preceding general specifications and a careful consideration of the particular needs of the linguist in writing translating programs, the following features are to be found in the COMIT system. The linguist can:

1) handle linguistic units of information without having to consider a fixed computer word length;
2) store and obtain access to information without having to compute addresses;
3) manipulate the information without consciously having to line it up and force it through an arithmetic unit;
4) add, delete, rearrange, and replace linguistic units without consciously having to set aside storage space for them;
5) add classificatory subscripts to linguistic units and cany out certain useful operations with these subscripts;
6) incorporate dictionary look-up operations;
7) introduce conditional rules and program branches in a simple and direct way;
8) write instructions in a natural and flexible format with a few carefully chosen abbreviations for compactness;
9) call the objects of interest to him by mnemonic names of his own choosing.

COMIT has two separate methods of addressing—one for instructions and one for data. This has the advantage that each method can be designed to be convenient for its purpose. The only possible disadvantage might be an inability to modify instructions, but COMIT has several built-in facilities for this.

The method of addressing instructions is similar to the method used in most programming systems except that there are no absolute addresses. The programmer uses symbolic addresses exclusively. The method of addressing data is quite different. Data is not stored by address, but is stored as a series of items called constituents in what amounts to one long expansible register called the workspace, fig. 1, and can be obtained by specifying in the

$$\underset{C}{\underline{\ldots.}} + \underset{C}{\underline{\ldots.}} + \underset{C}{\underline{\ldots\ldots..}} + \underset{C}{\underline{\ldots.}} + \underset{C}{\underline{\ldots\ldots..}} + \underset{C}{\underline{\ldots}}$$

Fig- 1. Constituents, C, in the workspace

instruction enough about the information or its context for the computer to be able to find it. The result is that the programmer never has to compute an address, although there is a facility that allows him to do so if he wants. Instructions in COMIT are called rules. Each rule may specify a number of complicated conditions and operations, and frequently is a complete loop in itself. The rules are punched on cards in a free format in which the only card position that has any special significance is the first column. The parts of the rule are separated by punctuation in such a way that each part may take up as many card columns as necessary. A rule may be hyphenated and allowed to extend onto as many additional cards as desired. Comments (in parentheses) may be freely placed within a rule and will be ignored by the computer.

A rule has five sections, fig. 2: the "name," the "left half," the "right half," the "routing," and the "go-to," each with its special functions. Every rule has a name and a go-to, and these are always the first and the last sections. The left half

$$\underset{\text{name}}{\underline{\ldots\ldots..}} \quad \underset{\text{left half}}{\underline{\ldots..}} \quad = \quad \underset{\text{right half}}{\underline{\ldots.}} \; // \; \underset{\text{routing}}{\underline{\ldots\ldots}} \quad \underset{\text{go-to}}{\underline{\ldots\ldots}}$$

Fig. 2. Format of a rule

and the right half are separated by an equal sign; the right half and the routing are separated by two fraction bars. Briefly, the functions of these five sections are as follows: The name section contains the symbolic address of the rule, or an asterisk if the rule needs no symbolic address. The left half effectively addresses those constituents in the workspace that are to be operated on. It does this by citing certain of their distinctive characteristics or of their environment. This causes the computer to search in the workspace from left to right, scanning over the various constituents until it comes to the first ones that adequately meet the description written in the left half. The right half specifies the operations that are to be carried out. These may involve addition, deletion, or rearrangement of constituents, or the addition, deletion, or alteration of subscripts on the constituents. The routing section of the rule controls input and output operations, controls special list or dictionary look-up operations, allows two or more constituents to be coalesced into one constituent or one constituent to be broken up into a number of constituents, and controls the facility called the dispatcher, which has the ability to control program branches on the basis of its interpretation of subscripts. In the go-to is written the name or symbolic address of the rule that is to be executed next, or else an asterisk which signifies that the following rule in the sequence is to be executed next.

5. Examples

A few examples of how the rules of COMIT can be used to program various operations will now be given. These examples have been chosen to illustrate some of the more important features of COMIT.

Let us assume, to begin with, that some English text has been brought into the workspace. The text is contained in the workspace in the form of a number of constituents, one for each word or punctuation mark. If it is desired to replace every occurrence of the words THE MAN IS OLD by THE OLD MAN, we must delete one word and rearrange the other three. The following rule will do it:

OLD-MAN THE + MAN + IS + OLD = 1 + 4 + 2
OLD-MAN

In this rule, the rule name or symbolic address has been chosen arbitrarily to be OLD-MAN. The constituents that are to be searched for are written in the left half after the rule name and before the equal sign. Plus signs are used as marks of punctuation to separate the constituents. The computer searches from the left end of the workspace and locates the first occurrence of THE MAN IS OLD. In the right half is written a string of numbers that represent which of the constituents are to be rewritten in the workspace and in what order. This rule needs no routing section. The go-to says that after the rule has been executed, control should be transferred back to the same rule again.

When the same rule is executed again, the computer will search again from the left end of the workspace, but this time it will find the second occurrence of THE MAN IS OLD because the first occurrence has been changed to THE OLD MAN, which the computer will skip over in its search.

The computer will break out of this cycle or loop only when all of the occurrences of THE MAN IS OLD in the workspace have been replaced by THE OLD MAN. When this happens, the search in the workspace initiated by the left half will be unsuccessful, the right half, routing, and go-to will not be executed, and control will be transferred automatically to the next rule. This is the first type of program branch available in COMIT: automatic transfer to the next rule if the structure represented in the left half cannot be found in the workspace.

If it is desired to reverse the process and replace every occurrence of THE OLD MAN by THE MAN IS OLD, we have to rearrange the constituents and add one new one. New constituents can be added simply by writing them in the desired place in the right half:

```
MAN-OLD   THE + OLD + MAN  =  1 + 3 + IS + 2
                                          MAN-OLD
```

The free format of the rule and the use of + and = as punctuation allow optional spaces to be inserted for increased readability. With all optional spaces eliminated, the preceding rule in its most compact form would read:

```
MAN-OLD  THE + OLD + MAN = l + 3 + IS + 2  MAN-OLD
```

Rules such as these may frequently be useful, but it is possible to write more general rules that have a wider range of applicability. There are two devices for this purpose. The first device makes use of subscripts; the second makes use of context. It is possible, for example, if it is linguistically appropriate, to place a subscript ADJ on all adjectives and define a noun in terms of its context as the word occurring between THE and IS.

The following rule will place an ADJ subscript on all occurrences of the word OLD:

```
OLD-SUB    OLD = 1/ADJ    OLD-SUB
```

(We assume for the moment that it is possible to recognize adjectives out of context.) In this rule we note that the subscript is separated from the rest of the constituent by a fraction bar. A number of rules of this type could serve to place the subscript ADJ on all adjectives. These rules would then be organized into a list or dictionary by a special facility that utilizes a rapid dictionary search procedure.

By making use of these ADJ subscripts as well as context, we can replace our OLD-MAN rule by a more general adjective-noun rule:

```
ADJ-NOUN  THE  +  $1  +  IS  +  $1/ADJ  =  1 +  4 + 2
                                          ADJ-NOUN
```

In this rule, the symbol $1 stands for any single constituent. ($2 would stand for two adjacent constituents, etc.) We do not use X for this because it might be confused with a letter in a word. The rule thus instructs the computer to search in the workspace for the first sequence of four constituents, where the first one is THE, the third one is IS, and the last one has an ADJ subscript on it. (It may have other subscripts too, but they will not interfere with the search at this point.)

As a further example of the utility of subscripts, suppose that we want the left half to find a genitive or a dative German noun phrase. In order to do this, German words in the workspace are first looked up and replaced by a part-of-speech symbol and a subscript, G-C, indicating gender and case. Each subscript G-C has associated with it the values that the gender-case variable may have for that word. For instance, DER would be replaced by ART/G-C M-NOM F-GEN F-DAT P-GEN and MUTTER would be replaced by NOUN/G-C F-NOM F-GEN F-DAT F-ACC. Then the left halves of either of the following rules would find these two constituents in the workspace:

```
A   ART/G-C F-GEN + NOUN/G-C F-GEN = ... OUT
B   ART/G-C F-DAT + NOUN/G-C F-DAT = ... OUT
```

In other words, if subscripts are mentioned in the left half, a constituent will be found in the workspace if there is an inclusion relation such that subscripts and values on the constituent in the workspace include the subscripts and values asked for in the left half.

Subscripts may be moved from one constituent to another. Suppose one has a subject and a verb in the workspace. The subject has a subscript for number, either singular or plural, and it is desired to move this subscript onto the verb in order to make it agree in number with the subject. In the workspace, then, we have either SUBJ/NO SI+ VERB or SUBJ/NO PL + VERB. The following rule will move the number subscript from SUBJ to VERB no matter whether it has a singular or a plural value.

```
AGREE    SUBJ + VERB = 1 + 2/NO*l    NEXT
```

That is, in order to carry over a subscript, one mentions the subscript in the right half, followed by an asterisk and a number that indicates which constituent has the subscript to be carried over. The subscript will be carried over with all of its values.

When subscripts are carried over onto a constituent that already has a subscript of this kind but with, perhaps, different values, the new values replace the old ones if they have no values in common. But if they do have values in common, the constituent is left with just the values that they have in common. For example, if the workspace had in it, as before, ART/G-C M-NOM F-GEN F-DAT P-GEN followed by NOUN/G-C F-NOM F-GEN F-DAT F-ACC, the following rule would leave both ART and NOUN with the subscript G-C F-GEN F-DAT; that is, with the values that the subscripts have in common:

```
COMBINE   ART + NOUN = 1/G-C*2  +  2/G-C*l     ON
```

In addition to the "logical" subscripts that we have been discussing, COMIT also has numerical subscripts available. With these subscripts it is possible to perform arithmetic operations, and to control program branches that depend on whether a numerical subscript in the workspace has a value greater than, less than, or equal to a value indicated in the left half of a rule.

Thus far we have used several characters with special meanings in the rule. The equal sign and two fraction bars separate sections of the rule, the plus sign separates constituents, the fraction bar is used before subscripts, $1 means a single constituent, the numbers 1, 2, 3, etc. written in the right half refer to constituents represented in the left half, parentheses are used to enclose comments that the computer is to ignore. Sometimes it is desirable to write these symbols in the workspace. To take an example from algebra, suppose one wants to replace A (B + C) by AB + AC. In order to represent the (, +, and) in the left half and not have the computer confuse them with the special punctuation use of these characters, they are preceded in the workspace by an asterisk. This asterisk is automatically added in input operations and removed in output operations. The rule would then be written:

```
EXPAND $1 + *( + $1 +-*+ + $1 + *) = 1 + 3 + 4 +
                                     1 + 5 EXPAND
```

Note that in this rule we have done something that we have not done before. A one has been indicated twice in the right half so that the single constituent found by the first $1 in the left half will be written in two different places in the workspace.

Another problem arises if we want to replace AB + AC by A (B + C). In order to be able to write a general rule for factoring, we want to be able to indicate somehow that one constituent is repeated twice and therefore should be factored out without having to specify what the constituent is. This can be done in COMIT by representing the first occurrence of the constituent by §l, and then representing its second occurrence by a number that refers to the first occurrence in much the same way as the numbers in the right half refer to constituents. Our rule then becomes:

```
FACTOR $1 + $1 + *+ + 1 + §1 =
                = 1 + *(+ 2 + 3 + 5 + *) FACTOR
```

The left half of this rule calls for a sequence of five constituents in which the third constituent is a plus sign and the fourth is the same as the first.

If we want to replace D SIN(F) by COS(F) D (F), where F is unrestricted and may be any arbitrary sequence of constituents, we use the notation $ to stand for this string. The rule for this is:

```
DIFF-SIN D + -SIN + *(+$+*} =
    = -COS + 3 + 4 + 5 + 1 + 3 + 4 + 5 DIFF-SIN
```

In this rule, besides the use of § to represent in the left half any number of constituents, we have used a hyphen to represent the character 'space' in the workspace. We shall now explain how the routing section controls input and output operations. The following rule will bring in a number of characters, one character at a time.

```
INPUT $ = 1 + X // *RAA2 INPUT
```

This rule places an X to the right of the constituents already in the workspace; the abbreviation *RAA followed by the number 2 then replaces the second constituent, namely the X, by the next character at the input. The rule will continue to bring in characters until there are no more characters at the input; then there is an automatic transfer of control to the next rule. It is also possible to bring in material a constituent at a time instead of a character at a time.

The output instructions are similar. It is possible to send to the output any specified constituents, or everything in the workspace. The following rule will write in the output everything in the workspace between the markers *A and *B:

```
WRITE  *A + $ + *B // *WAA2 CONTINUE
```

COMIT has a provision for address modification called the dispatcher. With the dispatcher it is possible to control an n-way program branch in a convenient manner. The program branch itself is set up in terms of a special kind of rule that can have a number of subrules. Whenever this rule is executed, the choice of which subrule is to be executed is determined by an entry in the dispatcher. Entries can be placed in the dispatcher by writing them in the routing section of any convenient rule. As an example of the use of the dispatcher and a program branch using a rule with subrules, consider a routine that brings in a number of characters from the input, and processes each one. In order to break out of the input loop we can use the automatic transfer of control to the next rule that occurs when there is no more material at the input. If one wants to enter the processing routine once more after this automatic transfer, a pre-set program branch can be used as in the following program:

```
INPUT     $ = 1 + X // *RAA2, STOP NO PROCESS
*                              // STOP YES
PROCESS  …
PROCESS  …              .
.        …              .
.        …              STOP
STOP NO                 INPUT
     YES                *
```

In this program, the dispatcher entries STOP NO or STOP YES are sent to the dispatcher from the routing section of either the input rule or the rule to which control is transferred when there is no more input. Then, each time the rules starting with PROCESS have been executed, control goes to the rule STOP. In this rule, control goes back to INPUT if the dispatcher contains the entry STOP NO. If the dispatcher contains the entry STOP YES, the second subrule is executed. This subrule has an asterisk in the go-to that transfers control to the next rule. Since there is no next rule, the program stops.

A rule may have as many as 36 subrules; if a larger number of branches than 36 is needed, several rules with subrules can be placed in cascade.

It is possible to indicate more than one subrule in a dispatcher entry. The dispatcher entry BRANCH A B D means execute in the rule BRANCH any one of the subrules A, B, or D. The choice is made at random.

When a dispatcher entry is sent to the dispatcher, it combines with the entry that may be there in the same way that subscripts combine when they are carried over onto a new constituent: the new one replaces the old one if there are no values in common, otherwise only the values in common remain. Subscripts themselves may be written in the form of dispatcher entries and sent to the dispatcher. By this mechanism, some rather complicated conditional transfers can be set up.

6. Conclusions

We have given a brief account of the factors that led up to the development of COMIT, and some of the characteristics of this programming language. It is too early to give an adequate evaluation of how it will work out in practice, but so far it has been a great help to linguists who have become familiar with it. Some trial translating programs have already been written in COMIT in anticipation of the availability of the compiler and interpretive routine. It appears that, although the system was developed specially for application to mechanical translation research, it may also be useful for other types of program involving the manipulation of nonnumerical symbols. Some of the types of program for which COMIT may turn out to be useful are: formal algebraic manipulations; compilation of programs from English or other more convenient descriptions; theorem solving, game playing, and learning programs.

7. References

[1] CHOMSKY, N.: *Syntactic structures.* 's-Gravenhage: Mouton & Co. 1957.

[2] YNGVE, V. H.: *A programming language for mechanical Translation.* Mechanical Translation 5, 1958, pp. 26—41.