G | Symposium on automatic programming

Coordinator A. J. Perlis, Carnegie Institute of Technology, Pittsburgh, Pa. (USA)

1. Introduction

by *A. J. Perlis*

Automatic programming (AP) is concerned with transforming problem oriented command languages into machine language codes. The solution of a problem generally begins with the statement of an algorithm which must be given in some notation, preferably one which provides full operational and descriptive power and uses an optimum number of symbols. The need to transform this into machine code itself leads to algorithms which should be written in a problem oriented language. This gives a recurrence procedure

and implies that a few problem-oriented languages are sufficient.
The first problem oriented languages (including ALGOL) have been designed for the algorithms of numerical mathematics. Their forms are essentially the same, though details often depend on the computer used. The only descriptions so far available of the translation processes are given in machine language partly because the problem is new, and partly because the algorithms are more complicated than those for which ALGOL was designed. The algorithms are in fact close to those needed for translating natural languages and an AP language to describe them would require:

1) Organization of data into lists whose structure and size cannot be specified a priori;
2) Recursively defined functions whose domains may include data organized as in (1);
3) Specification of functions as sets over which a data dependent priority scheme operates as a selection principle c.f. Markov's definition of algorithms;
4) Dynamic generation of algorithms;
5) Integer arithmetic.

In addition there are code translation processes which arise when algorithms are combined, for example the construction of assembly systems which is one of the most difficult tasks in AP. There are problems associated with the reduction of unnecessary duplication for entering subroutines and with the specification of unambiguous procedures for communicating information to subroutines.
Finally, AP systems should permit dynamic intervention, possibly using checking circuits, for example when critical situations of low probability arise. Attention must also be given to diagnostic procedures to allow for program testing when there is a translation between the original program and the machine code realisation.

2. Code to code translation systems

by *A. W. Holt (USA) (read by A. J. Perlis)*

Present automatic programming systems, while of great value, have shortcomings which are due to the basic approach employed. They are usually prepared by the computer manufacturer, and once written they are almost as difficult to change as the computer itself. The individual user rarely produces his own system, because the process is long and costly.
Work with UNIVAC I and II has shown that the translation program should not be written directly in machine code, but in terms of auxiliary programmed components. If such components were supplied by the manufacturer, users could develop their own systems as they obtain experience in their particular problems. It also becomes possible to translate many problem codes into a particular machine code, or vice versa.
Analysis of the translation process has led to the division of the UNIVAC system into a "system core" containing a few basic components and a "translating library" whose components are used with varying frequency. Examples are:

A) Core functions:
   1) Overall control and the determination of priority amongst functions to be performed;
   2) Services such as
      a) Dictionary look up
      b) Contextual and absolute substitutions
      c) Determining class membership
      d) Determining order relations
   3) Address interpretation, allowing chains of indirect reference for assembly.

B) Library functions:
   1) Decoding parenthesis structure into a set of priority numbers
   2) Providing translating dictionaries
   3) Economising the assembled code
   4) Applying formal rules to strings of digits in certain machine instructions

Translations typically proceed by a chain of steps in each of which a core element is present, controlling one or more library elements. The problem is given initially in problem code and must be reduced to computer code. The library elements are largely determined by the problem code in the early steps and by the machine code in the later steps, while the same core element may be used in many steps.

3. A variation of the ALGOL language

by *H. D. Huskey (USA)*

Some modifications to ALGOL are proposed, with particular attention to the problems of using the language with a small computer and of expressing mathematical formulae. There is no change in the symbols used, but the logical relations of some formulae are changed. A basic limitation to the attempt to use normal mathematical notation arises from the need to write all symbols at a single level in each line, i.e. to avoid superscripts and subscripts.
ALGOL is extended by defining the value of an expression E as a sequence of bits with particular values 1,0 if all the bits are 1 or 0 respectively. An expression E_1 may have a value other than 0 or 1 and may then be used in the expression

$$E_1 \wedge V = : X$$

which allows the use of E_1 as a mask to be applied to V.
Another extension is the form of conditional statements, namely

$$A = B : A = : D$$
$$C = : D,$$

which means that if A = B then A replaces D but if A ≠ B then C replaces D. This may be placed on two lines for the sake of legibility in the manuscript.
Finally there is the problem of dimensioning. Declarations are added in the form

$$, A(5),$$

which states that there are five items A_1, $A_2 \cdots A_5$. Similarly there may be multi-dimensional declarations, such as

$$, A(10, 25),$$

Such statements may be distinguished from subscripted variables by the punctuation marks which appear on each side. This may be used to form statements, such as the following for matrix multiplication

```
S1 : a(10,20), b(20,10), c(10,10)
     [[0 =: c[i,j], [c[i,j] + a[i,k] × b[k,j]
        =: c[i,j]] k = 1(1) 20] c = 1(1) 10] j = 1(1) 10
```

4. Algorithms and machine logic for analysing economic activity

by *N. E. Kobrinski (USSR)*

1) The analysis of the activity and management of an industrial enterprise is based on the study of information obtained both in the course of manufacturing processes and from outside. As a rule, the information "word" comprises two elements: the classification group and the base. The first defines the economic attributes, while the second is a concrete or an abstract number.

2) Accounting and planning algorithms are obtained by means of complex operations which involve both the classification groups and bases. These operations require a comparatively small set of standard operators.
3) The standardization of operators allows the design and characteristics of the machines for economic estimation and calculation to be optimised taking into consideration the scope of processed information, the predetermined form for the results and the reliability, cost and service conditions of the machine.
4) The realization of the above points is illustrated in a machine intended for economic analysis, which is at present under development.
5) The economic efficiency of a machine is determined not only by the reduction of labour expended in processing information but also by the reduction of the time lag in the data processing circuit.
6) Further development of means of automation for management control at manufacturing plants demands the use of mathematical schemes which are adequate for actual management systems, and of algorithms which ensure the realization of an optimum management process. It is also necessary to proceed with further investigations on the economic efficiency of machines intended for the automatic management control of industrial plants.

5. Autoprogrammation pour Gamma 60

par *Mme. J. Poyen (France)*

L'auteur a présenté une déscription précise du langage et de la syntaxe A.P.3, et un bref aperçu des possibilités de ce langage. A.P.3 est une programmation automatique de type algébrique, destinée à l'écriture de problème scientifiques, et conçue à l'usage du calculateur Gamma 60 de la Compagnie de Machines Bull. Elle est assez proche du langage courant pour que tout lecteur familier avec les expressions mathématiques puisse comprendre d'emblée la signification de tout programme. On peut inclure dans la bibliothèque de sous-programmes d'A.P.3, tous les sous-programmes ou programmes existants dans un bureau de calcul, qu'ils soient écrits en langage machine, avec un code symbolique moins évolué ou avec l'A.P.3 lui-même.
Les éléments constitutifs des programmes d'A.P.3 sont l'alphabet latin majuscule et minuscule, les chiffres arabes et les signes d'opération. Un problème se présente sous la forme de phrases, composées à l'aide des symboles définés précédement. Ces symboles peuvent êtres groupés en mots. Certains de ces mots sont les mots «clés» et les mots complémentaires qui constituent «l'ossalure» de la phrase. D'autres sont utilisés pour désigner les opérandes et les sous-programmes enregistrés.
La nature des opérandes dans un programme A.P.3 est précisée par une phrase qui utilise le mot clé «Definition». Les opérandes peuvent être

des nombres réels en simple ou double précision
des nombres complexes en simple précision
des matrices, vecteurs, etc.
des indices
des quantités indicées

On écrit un formule sous la même forme que les opérandes scient des nombres réels ou complexes, des matrices, etc. Aucune précision n'est nécessaire quant à la dimension des opérandes utilisés. Elle ne sera indiquée qu'au moment de l'exploitation.
Les autres mots clés précisent par exemple les ordres de saut, de commande à distance, de préparation, d'entré et de sortie, et les ordres sous-programmes. On peut se servir de ces mots pour commander une organisation quelconque du programme.
Pour la mise au point du programme A.P.3, l'utilisateur a à son disposition des outils puissants pour la détection et la correction des erreurs. En cas de correction seule la phrase erronée doit être réintroduite et non l'ensemble du problème.

6. *H. Riesel (Sweden)*

The "alpha code" system for the Swedish computers BESK and FACIT has certain features which have not been mentioned here. A facility "order control" causes the program to punch out sufficient information to identify the alpha code orders being obeyed. This allows the diagnosis of obscure errors in a new program. Similarly if a forbidden operation is attempted, sufficient information is printed to identify the fault.
The system includes a means for compiling routines written in alpha code which guards against a "collision" in the names given to routines by different programmers. Finally there are orders for automatic routing of instructions and data which are of great value with a small high speed store.

7. The cellar principle for formula translation

by *F. L. Bauer* and *K. Samelson (Fed. Rep. of Germany)*

Within the framework of an algorithmic formula language such as ALGOL there are statements of very different complexity relative to the machine code. Some are practically identical to single machine instructions and need not concern us further. On the other hand, a real problem is posed by bracket structures which cannot simply be evaluated sequentially in terms of machine instructions, of which arithmetic expressions are an important case. Rutishauser proposed a translation method starting with the innermost brackets-pair, which however requires multiple, or at least back and forward, reading of the formula program.
Brackets and addition symbols separate terms which can be evaluated individually. It follows that multiple scanning of the formula program can be replaced by symbolwise sequential evaluation of formulae and an auxiliary store called the 'cellar'. Every symbol appearing is either evaluated immediately or, if this is not possible, it is stored in the cellar which works on a strict 'last in — first out' basis. Only the symbol entered last into the cellar is needed at each stage of the translation process and this symbol in conjunction with the next symbol of the formula determines the actions of the translator.
Disregarding some special cases for the sake of simplicity, numbers and identifiers for variables can be evaluated immediately in terms of addresses. The first symbol (except numbers and variables) occurring in an arithmetical expression is entered into the cellar automatically and subsequent symbols are always tested against the uppermost cellar symbol. If the new symbol is not higher in precedence than the cellar symbol, the cellar symbol is removed from the cellar, and the corresponding machine instructions are attached to the sequence forming the program. When the new symbol is lower in precedence than the cellar symbol, the comparison process is repeated with the next cellar symbol. Finally, the new symbol is entered into the highest cellar position. An opening bracket indicates the necessity to store away an intermediate result. The opening bracket is entered into the cellar, and is cancelled out only by the corresponding closing bracket. Assignment of storage locations for intermediate results can be handled on a similar 'last in—first out' basis in a 'numbers cellar'.
Admission of functions and subscripted variables requires only slight extensions of the cellar principle as long as the value of the storage function of a subscripted variable is to

be computed directly and completely every time it is needed. Since in the general case this requires multiplications, it is time consuming, and cannot be tolerated for running subscripts in inner loops, where functions must be evaluated recursively using additions only. For general polynomial subcript expressions, this becomes rather complicated. Therefore, recursive evaluation has been generally restricted to subscript expressions which are linear in the respective running variable. This means that the translator must test for linearity, and provide for calculation of initial value and increment of the storage function of the subscripted variable outside the loop, and for addition of the increment by means of lines or otherwise inside the loop. Furthermore, identity of increments for different subscripted variables should be established for efficient use of B-lines. The initial value and increment of the storage function may depend on other running variables belonging to exterior loops. In this case, the translator may again provide for recursive calculation.

8. General Discussion

S. Gorn (USA): With present techniques it may take 30 man years to write a compiler (cf. FORTRAN). With techniques now under development it may take 3 man years while in five years' time it may require only 3 man months.

J. W. Carr III (USA): Compilers can be written for a new machine with the aid of a compiler working on an existing machine, though ALGOL would need some additional facilities for this purpose. In this way no detailed machine language coding is necessary.

J. W. Backus (USA): Much of the 30 man years quoted by Mr. Gorn for FORTRAN was spent in examining the problem to obtain an efficient compiler. The use of a compiler to build a compiler is unlikely to give an efficient program and so a lot of time will still need to be spent to this end.

J. V. Garwick (Norway): Small machines demand special techniques for compilers, including ALGOL compilers. It may be necessary to punch out the program at an intermediate stage using floating addresses, and to feed this information to the machine again at a later stage.

M. Landau (Fed. Rep. of Germany): The problem of treating multi-dimensional arrays may be eased by the use of more elaborate modification systems.

A. Caracciolo (Italy): A machine has been built with double modification, but it has been found that this has not significantly eased programming problems.

K. Samelson: It is doubtful if multiple modifiers can help in the treatment of arrays unless the size of the array is a power of the radix used in the machine. Only a novel of storage could really help.

H. J. Maehly (USA): In the past, machines have been built originally for "direct programming". It was found out later that this was too tedious and auxiliary routines—interpreters, compilers, etc.—have then been written. This is not very efficient. Instead, we should either construct machines for which direct coding is extremely easy and simple or, as this seems almost impossible, plan our machines not only for fast computing, but also for fast and efficient code translation and compiling. In other words, the logical design of the machine proper and of its translating routines should be considered as an entity.
The second remark concerns the influence of automatic programming on education. Program-oriented languages will conceal details of computing efficiency and problems of numerical stability. This has already been experienced with floating point arithmetics. It is certainly wrong and misleading to say that automatic coding or the use of ALGOL could replace mathematical training. On the contrary, the purpose of automatic coding is mainly to allow the well-trained scientist and engineer to program for himself, without undue loss of time. This will be successful if and only if potential users of computing machines and of ALGOL translators are well trained in numerical analysis.

A. J. Mayne (UK): Present developments in automatic programming still leave a need for many programmers, particularly for commercial and economic uses of computers. There is also a requirement for mathematicians who can classify computer problems into types which will each use a special language. In this language, any redundancy in the original statement of the problem will be avoided and a short statement will result which can be used as the basis of a program. Important problem types arise in statistics and some classification has already been attempted at the Rothamsted Experimental Station.
There are ways in which programmers can still exercise ingenuity, particularly in the design of autocodes and in problems requiring continued modification of orders. It is probable that in future programmers will have to be more intelligent than at present whilst autocode users will need much less detailed knowledge of the machines than contemporary programmers.

K. Samelson: AP does not affect the need for mathematical analysis of the problem, and programmers must be educated to do this rather than to be coders. They should be enabled to write programs in a simple code and to use a computer as a code transformer.

H. D. Huskey: Most great advances in mathematics have been made possible by improvements to notation. There is a need for improvement in the notation for AP, particularly a requirement for a simple notation for simple actions.

R. W. Bemer (USA): In any AP system care should be taken that the translation time does not exceed the program running time. This may involve a preliminary translation from an external to an internal symbol, i.e. from a natural language code word to a machine code number, particularly for commercial application.

H. J. Maehly: AP and data processing will require similar computers. There is a need for an intermediate AP language to ease translation problems and to allow the re-use of parts of compilers for different applications.