CURRENT PROBLEMS IN AUTOMATIC PROGRAMMING

Ascher Opler

Computer Usage Company, Inc.

18 East 41st Street

New York 17, New York

Summary

The proliferation of the number of applications to be programmed and the number of types of computers available has created a significant and challenging problem. This survey will consider some of the more promising suggestions for enabling automatic programming to keep pace with other developments in the field. Among the prospects are: standardization of source languages, development of a standard universal intermediate language, design of computers to operate directly in source language, automatic translation of programs between computers, the automatic construction of compilers and the standardization and construction of common compiler modules.

Automatic programming has grown from an interesting child to a troublesome adolescent in the past few years. The number of publications and meetings devoted to its problems has continued to increase. This presentation will attempt to state the problems from a practical viewpoint and to survey a number of techniques that have been proposed to handle some of the current difficulties. The viewpoint taken here is a practical one and considerations of costs, staff, delivery time, competitive position, etc., will not be foreign to the discussion.

The number of groups developing automatic programming (and closely related systems) today is enormous and continually increasing. Personnel so involved are primarily (a) representatives of the computer manufacturer, (b) system programmers at the more advanced computer installations and (c) those involved in computer programming research (most frequently at universities and large independent research organizations).

With time, the responsibility for developing system programs has shifted from the user to cooperative groups and now to the manufacturer. Computer manufacturers are expected by their customers to supply with the machine a complete set of automatic programming systems. With the field becoming highly competitive, each manufacturer is expected to deliver the automatic programming package at the same time the computer is delivered. Manufacturer's representatives are asked questions concerning the programming package and its availability more often than they are asked questions regarding the hardware and its availability.

Dependence upon the applied programming staff of a manufacturer is not completely universal. A dozen or so highly experienced computing equipment users have developed their own systems either because they believe them to be superior to those of the manufacturer, more compatible with their own operating methods or can be made operational months before equivalent programming systems promised by the manufacturer. The achievements of this group has been particularly impressive.

Most automatic programming systems built by manufacturers have been based on fairly solid concepts developed over the past few years. Fortunately, there are a group of brilliant energetic researchers in the automatic programming area who have been making rapid strides toward developing greatly improved techniques. Some of these contributions will be reviewed in the light of the overwhelming problems confronting the programming field today.

Mounting Pressures on Developers of Automatic Programming Systems

Those who would develop automatic programming systems for production purposes are faced with many pressures. Some of the most troublesome are outlined below.

A. Increased Requirements

At the present time, the automatic programming systems supplied by a manufacturer (and expected by his customers), fall into two categories: (1) compilers and (2) other systems

(including assemblers, operating and debugging systems, sort generators, etc.). For most of what follows, the discussion will center on the compilers (defined as programs which translate from a source language (usually problem oriented) to a machine language). In the area of automatic programming systems, the manufacturer is expected to supply compilers to translate from one or more standard algebraic languages and from one or more standard commercial languages. Furthermore, these compilers should be able to operate effectively on any of the numerous configurations of equipment that one may order and install and to produce object programs that are also effectively operable on any of this wide degree of equipment modularity. It is not uncommon for the purchaser of equipment to request assurance that the manufacturer will supply compilers to translate the same language for all successor machines that he may market in the future. A large manufacturer may very well have as many as ten compiler projects going simultaneously to prepare translators for several current, several announced but undelivered and perhaps one or two unannounced machines.

These requirements for "general purpose" languages for "general purpose" computers also carry over into the area of special purpose languages (automatic programming for numerically controlled machine tools; natural language translation; special military task programming, etc.) and into special purpose computers. At the present time one might hazard a guess that the amount of automatic programming effort in the category of the "special purpose" areas is approximately equal to that in the "general purpose" areas.

B. Shortage of System Programmers

The task of constructing automatic programming systems is generally relegated to the direction of those familiar with the art. Up to the present time, it has been a highly specialized skill and those who practice it are much in demand at premium salaries. The number of competent experienced system programmers is very few while the demand is very high. There is also at present very little concerted effort to train for this specialized programming area.

C. Mounting Costs of Automatic Programming

At the present time, the development of a compiler will range from less than $100,000 to over $1,000,000. These costs include the man months spent in programming and check-out, large quantities of machine time, great documentation efforts, education, training manuals, etc.

D. Developmental Time

The time to develop a compiler today will vary from six months to better than two years depending upon the type of compiler required, its quality and its associated features (diagnostic system, restarts, compatibility, modularity, etc.).

E. Simultaneous Development of Computer and Compiler

Because of the demand by customers that manufacturers deliver compilers starting with the first machine, an added pressure is placed upon the supplier. He must program and check out the compiler during the period of construction and testing of the computer prototype. Therefore, the compiler work is hampered by the necessity of using large amounts of simulation on another computer, working on an engineering model and operating without the availability of programming and debugging systems which are being concurrently developed. Considerable difficulty will also arise because of the large quantity of machine time required for effective check-out and testing of a compiler.

It is easy to see, under the extreme pressures created by the competitive marketing situation (which itself results from the rapid acceptance of automatic programming) and the continual spread of computer technology into new areas, that the pressures on developers of automatic programming systems are enormous. It is little wonder that they are rapidly scanning the horizon looking for developments that will reduce their costs and enable them to deliver new automatic programming systems rapidly and with a limited staff of system programmers.

Let us turn our attention now to some of the solutions that have been offered. These are divided into three areas: Elimination of the necessity of writing compilers, minimization of the number of compilers to be written, and minimization of the effort of writing each compiler.

Elimination of the Necessity of Writing Compilers

Recalling the well-known equivalence of hardware and programming leads one to consider the possibility of designing computers that will accept and directly execute programs written in source languages. Thus, in effect, the

burden of translation would be eliminated and the source language would be the machine language.

From the fundamental nature of a Turing machine, one may deduce that, while the crude input could scarcely be executed directly, it is feasible to automatically convert the input to some directly useable form by programming. To pursue this further, consider a program available for the IBM 1620 that is called "GOTRAN". This program accepts FORTRAN statements and produces as output (in a single manipulation) the results of executing the statements. If one had this computer with the "GOTRAN" program locked-in, then he could consider it to be a machine that directly accepts and executes source language. However, this is really chimerical because the actual effort of writing "GOTRAN" was certainly of the order of magnitude of writing a compiler and thus we have actually eliminated no programming effort. What has been accomplished is the "conversion" of a general purpose computer to a special purpose device via programming.

For the achievement of the equivalent of programmed source language operation but with elimination of much of the programming, one would have to move closer to hardware developments. Two approaches that offer promise are the use of micro-programming and the design of computers with compiling requirements as a primary criterion.

The use of micro-programming has been known and discussed for some time. We hear much more today about the "customized" computer that can be micro-programmed to order. With a micro-programmed machine, it might well be possible to build aggregates of micro-steps that would carry out compiler instructions. Thus, one step toward the computer that would process source statements directly is the avoidance of construction of any logic higher than a micro-program step. The same question now appears - is the effort of micro-programming the machine to accept source language any less than that of constructing a compiler for a general purpose machine?

The other approach, to be described in the paper by R. S. Barton later in this session, lies closer to computer design. Ultimately, logical designers and automatic programming experts will be forced to work together.

Minimization of the Number of Compilers Required

A. Standardization of Source Languages

One effort that has been proceeding for the last few years and apparently producing considerable success is the cooperative movement to standardize source languages. There has been considerable international effort in the development of ALGOL and considerable national cooperation in standardizing on COBOL. Standardization will certainly continue in these areas. Ultimately, the fate of standardized languages will depend upon their acceptance. No matter how many agencies endorse a standard language, if the user finds a non-standard language compiler available which meets his needs, he will make no effort to use the standard language. Perhaps standard langauges cannot be accepted until non-standard languages are banned or plans are made to phase them out over a period of time. An interesting paper will be presented by Miss J. Sammet at this session suggesting even further reduction in the number of standard source languages.

B. Use of a Standard Intermediate Language

A proposal was made several years ago to interpose between the problem oriented languages and the computer oriented languages a universal computer oriented language (UNCOL). In a greatly oversimplified manner, if there are n problem oriented languages and m computer oriented languages, it requires (n x m) translators to translate from each POL to each COL. If one goes through the intermediate step and writes n translators from POL to UNCOL and m translators from UNCOL to each COL, then one replaces (n x m) translators with (n + m) translators. The present status of this development will be reported in a paper in this session by Mr. Thomas Steel.

C. Solutions to the Component Modularity Problem

With modern computers available as a collection of modules (of internal memories of various sizes, various input/output, buffering and control systems, magnetic drums, tape, discs, etc.), it becomes increasingly difficult to handle the two configuration problems, the compiling and the object complement of equipment. Thus far solutions offered to this general problem have not been common. Holt and Turanski have discussed an Allocation Interpreter for the object configuration problem. The general

solution for the compiling configuration problem has been the simple change of table sizes and buffer sizes (thus speeding and expanding compilation) to adjust to the internal storage requirements during compilation. The answer to this problem when dealing with hierarchical memories with different access methods and speeds is sorely needed. The answer frequently offered by the manufacturer is to arbitrarily waive all but one or two compiling and object configurations. It is hoped that techniques will be forthcoming in the next few years that will allow each installation to make optimum use of whatever computing equipment is available to them.

D. Automatic Translation Between Source Languages

This is a possibility only in the special case where one source language is (or can be converted to) a subset of another. Fortunately, the FORTRAN language (which has become so common on many existing computers) is amenable to translation to ALGOL. At the present time, several FORTRAN-to-ALGOL translators are being written. This may also appear to be the relation between several data processing languages and COBOL.

E. Automatic Translation Between Object Languages

This is an area that is now being quite actively investigated. At one time, it was believed that the only method of source language to source language translation was by means of interpretive simulation. A considerable amount of research has been carried out recently in re-examining the possibility of machine language to machine language translation at a higher level and preliminary reports are encouraging.

Minimization of the Effort of Compiler Writing

A. Compilers Capable of Writing Other Compilers

This has been the subject of a tremendous effort in the past few years. When the idea of automatic programming was first proposed, one of the various suggestions was that, by "bootstrapping" with one operating compiler, one would be able to construct compilers for other machines, for other languages, and in fact even better compilers than the original one. While it is perhaps not universally agreed upon, this has been, in general, a disappointment. It has been repeatedly demonstrated that one can use a compiler to write a compiler, but the quality of the compilers so produced, either in terms of the time required to compile, the memory space requirements or the quality of the object program has been such that these demonstrations have been primarily of academic interest. There has been a tendency to design special purpose source languages whose primary function is the description of the manipulations used in compiling. These have been somewhat more successful. Furthermore, the efforts of the compilers in automatically writing compilers have been primarily limited to the writing of algebraic compilers. It will appear that in the next few years, there will be more need of compilers for the production of data processing compilers than for algebraic compilers. There is every reason to believe that continuing developments in this area, particularly in that of improving the language of a compiler writing compiler will bear fruit before too long.

B. Develop Special Compiler Writing Systems

When one wants to produce a compiler automatically, the circumstances are generally such that no compiler of the type required already exists for the machine in question. Thus the compiler of compilers mentioned above usually operates on a machine foreign to the one for which a compiler is needed and must go through a complicated conversion and bootstrapping routine after compilation of the compiler. A proposal has been to develop a large system in which the inputs are the desired language for the compiler to be produced and the description of the machine on which the compiler is to operate. This is the concept of the SLANG system under development by R. A. Sibley of IBM.

C. Develop Techniques to Facilitate Compiler Writing

At the present time, the writing of a first class compiler is entirely dependent upon the careful fashioning of all sections of the compiler by manual programming symbolic language, instruction-by-instruction. The quality compiler of today will require from 25,000 to 50,000 machine instructions each of which must be hand written and debugged. It has been obvious for some time that, if techniques are available to reduce the amount of coding to be done, it will greatly reduce the compiler writing effort. That is, since the complete elimination of hand coding by using a compiler of compilers is not satisfactory, what is needed is a collection of tech-

niques for minimizing the effort. Let us consider what programming aids have been offered.

1. List and String Processors

Since the nature of most source languages is a rather free running string of meaningful symbols loosely resembling English or mathematical notation, one important task in compiling is the analysis of these strings into suitable origins, delimiters, and terminators and the isolation of the included identifiers resulting (after recognition of their contents) in construction of tables or codes. This suggests that an improved scanning and recognition process would be extremely useful. Among available techniques are the threaded list system of Perlis and associates, the Newell-Simon-Shaw list-processing approach and others.

2. Generalized Analyzers and Generators

Following the scanning and decoding of the raw input, the scanned input is analyzed for its meaning in terms of the whole source program. Holt and Turanski have pointed out that, for a given language, the programming of both the scanning and the analysis is virtually identical no matter what the object machine and the nature of the object program will be. They have therefore suggested the stockpiling of either entire analysis sections or of the logic of analysis sections and making effective use of these in all compilers originating from the same source language. For the actual synthesis or generation of object program, a number of algorithms for producing code from algebraic languages have been developed and published. For the data processing compilers like COBOL, it is customary to use "generators" which produce the desired object program sections. Currently there is fruitful development in the area of generalizing these generators so that they can be tailored to produce desired object coding without re-developing the whole generator for each compiler.

3. Macro Instruction Systems

Another aid to the compiler builder is the use of the most advanced type of macro instructions. It is well known that as assembly programs become more sophisticated and comprehensive, the requirements of the compiler writer become less and less. This may be seen in the ease with which compilers may be written where highly sophisticated assemblers are available. The ALTAC compiler (from FORTRAN language to TAC (assembly) language) for the Philco S-2000 is a good example of this use. Even more advanced macro systems are MICA developed by Owen Mock of North American Aviation and MACROSAP developed by M. D. McIlroy of Bell Telephone Laboratories. An ultimate extension of this concept has been in the development of MOBL by North American Aviation which is a data processing language built entirely of macro instructions which are processed by the MICA system. Using the macro instructions concept, they were able to produce a compiler with a minimum of time and manpower that is capable of translating an excellent data processing language into 7090 symbolic language.

As we have seen, the pressures placed upon those with responsibility in the automatic programming area are enormous. With necessity the mother of invention, it is sincerely hoped that at least some of the many solutions currently being proffered will bring the chaotic situation in automatic programming under control by 1963 or 1964.