MACHINE INDEPENDENCE IN COMPILING*

Harry D. Huskey

University of California
Berkeley, California, USA

Since 1958, there has been a substantial interest in the development of problem-oriented languages as a means of controlling computers or data processing systems. All of these efforts have had as a primary purpose the goal of reducing the human effort necessary to prepare a problem for computation or processing on such a computing system.

Perhaps the most significant of these developments is the publication of the international algorithmic language called ALGOL in 1958 and a revised version in 1960. ALGOL can be described as a very general scientific language suitable for scientific or engineering computation. It has two purposes: one of these is for the communication or the publication of algorithms for solving problems, and the other is its direct use as an input language to computing machines. It has perhaps been more successful in the first case than it has in the second. However, there is a substantial number of translators which will accept ALGOL statements and produce machine language for the appropriate computer. In all cases, however, compromises have been made with the complete language. In some cases, these are very minor compromises; in other cases, they are very extensive compromises. Perhaps a more significant aspect of the development of ALGOL is the effect it is expected to have on the future design of computing machines.

Another language development, which has been sponsored by the Department of Defense of the United States, is that of COBOL, a common business-oriented language. In the meantime, the

*This research was supported by the Bell Telephone Laboratories under Grant D-603513 and by the Air Force Office of Scientific Research of the Office of Aerospace Research; the Department of the Army, Army Research Office; and the Department of the Navy, Office of Naval Research under Grant AF-AFOSR-62-340.

problem-oriented language which has been used most widely is the FORTRAN system, developed by the International Business Machines Corporation.

Simultaneous with the development of ALGOL 58, the author participated in the development of a problem-oriented language with a restricted field of application, namely that of simulation. In this particular problem there was no need for the various general features of ALGOL so a restricted language called NELIAC was developed and put into operation. This activity took place at the U.S. Naval Electronics Laboratory at San Diego. The unique feature of the NELIAC development has been that the translating system has always been written in the problem-oriented language itself. Consequently, revisions were easy to make and the description of the compiler on punched cards or punched tape was the up-to-date documentation of the system. Since this system was developed simultaneously with the development of the specifications of ALGOL 58, some of its features are more similar to ALGOL 58 than to ALGOL 60.

THE NELIAC SYSTEM

In the NELIAC system, it is possible to modify the command generators so that they will generate commands for a machine B on a computing machine A.

The original NELIAC system was written for a military computer called the Sperry-Rand M-460, and the first version generated commands for the same computer. Early in the development, the system which ran on the M-460 was modified so as to generate commands for the Datatron 205 and also for the Datatron 220. In another effort, the M-460 system was modified to generate commands for the CDC 1604.

With some help in the way of hand operations, the whole translating system was transformed from the M-460 to the Datatron 220 to obtain a NELIAC system which runs on the Datatron 220 and generates commands for the 220. The need for the hand operations arose primarily from the fact that the Datatron 220 is a decimal machine without access to binary bits, whereas the M-460 is a strictly binary machine. In a similar way, the system was transformed onto the CDC 1604 so as to have a NELIAC system which would generate CDC 1604 commands on that computer.

In another effort, the NELIAC system on the M-460 was modi-

fied so as to generate IBM 709 commands, and in a boot-strap type operation, the whole compiler was transformed onto the IBM 709 computer. A version of this was developed which generated IBM 704 commands, and of course the 709 version runs on the 7090. Various features have been added to the 7090 system to take care of the special machine features available.

In each of these cases, the NELIAC system is written in its source language form and can be recompiled on the specific computer. Consequently, if some variation in the translator is desired for a particular purpose, this is easily accomplished with a minimum of man-hour effort.

In all this development, however, whenever a new machine is considered, it is necessary to change the command generators so that they will generate commands for the new machine. If the logic of the new machine is different, or if there are features which are of sufficient interest to be used, then perhaps even the logic of the compiling has to be modified to some extent. Thus, it may require as much effort as six man-months to establish a NELIAC system on a new computer.

TRANSFERRING A COMPILER TO A NEW COMPUTER

Therefore, since new computers are going to replace old computers and it is very important that substantial programming efforts do not have to be done over again, it is of very considerable interest to be able to transform a compiling system onto a new computer without modifying the source language so that the body of the existing programming can be transferred onto the new computer as well. Programs written in ALGOL satisfy the restriction that they do not have to be revised in order to run on a new computer; however, all the current translators would have to be done again for any new computer. Therefore, one of the purposes of the activities described in this paper has been to minimize the man-hour effort required to transform a translating system onto a new computer.

In order to do this, the translating system has been conceptually divided into three major parts. The first part is called a preprocessor; the second part is called a translator; and the third part is called an assembly program. The preprocessor effectively takes care of identifiers and a few other features of the source language in such a way as to minimize the activities of the translator and the assembly program as far as identifiers

are concerned. Thus, one could say that the preprocessor essentially converts general identifiers into relative addresses in an appropriate computer. Along with these relative addresses certain flag information is established, indicating for example whether the identifier is a label, or relates to real, integer or Boolean-type quantities. Thus, the assembly program can quickly compile a correct object machine program; the preprocessor makes almost no changes in the syntax of the source program. The translating portion of the system accepts the output of the preprocessor and effectively passes the identifiers on through without modification. The syntax of the statements, however, is processed completely so as to obtain a sequential list of operations appropriate for a computer in any of a large number of types. In other words, the output of the translator might be said to be a type of Polish string, in which identifiers and operations are developed in the order in which they must be considered in the object computer. The output of the translator is called an *intermediate language.*

CHARACTERISTICS OF THE INTERMEDIATE LANGUAGE

It is clear that if the intermediate language is to serve for a number of computers, then it must be as general as the most general of any of the machine languages of the object computer. Another way to state this is that any information present in the source language about the problem which can be used in assembling the final machine language must be carried over to the intermediate language. For example, if the intermediate language referred to a single accumulator, whereas one of the object accumulators had many accumulators, it would be difficult to write an assembly program which could analyze the intermediate language and decide how to distribute the activity among the several accumulators. Consequently, such information must be developed in the preprocessor and the translator, and must survive in the intermediate language form of the program. In the same way, information about index registers must not be collapsed in any way so that if a sufficient number of index registers are present in one of the object computers, then indices can be handled in an efficient way to solve the problem.

Consequently, the intermediate language is set up on the basis that there may be an infinite memory, that there may be infinitely many index registers, that there may be, in fact, infinitely

many accumulators. In order to allow for extensions of the source language, there is provision for flagging the identifiers in the preprocessor so that the assembly program can be expanded to handle new types of identifiers. In this way more classes of variables may be considered than have been treated in the various problem-oriented languages. This "no limit" restriction on index registers and accumulators requires the possibility of variable length identifiers in the intermediate language. Consequently, simple identifiers, or ones that occur frequently in source problems, are represented in simple form, and as more and more variables are needed, then more lengthy identifiers may be used. Also, since this intermediate language must be machine independent, it is not feasible to talk about words. Thus the intermediate language deals with characters and groups of characters called identifiers, operators or flags. On the other hand, in order to obtain efficient operation, those symbols and operators which occur frequently are simply represented, and the first identifiers that appear in a problem are likewise simply represented. As the list gets longer, provision is made for more complicated representation for the identifiers.

THE ASSEMBLY PROGRAM

The assembly program takes the output of the translator, or the so-called intermediate language, and transforms this into absolute machine code for a particular object computer. Consequently, a different assembly program is required for each computer.

The identifier which come from the earlier stages of the translation process are essentially in relative address form. In such cases the assembly program simply adds a base address to these identifiers and uses this as the address for the appropriate object computer. General sequences of arithmetic operations which appear as output of the translator must be transformed into the appropriate machine language commands for the object computer. In some cases the command specified in the intermediate language may not correspond to simple one-command operations in the object computer. For example, the variables being processed may be real and the arithmetic operations need to be floating point, whereas the object computer mat be a fixed point machine. In this case the assembly program develops the appropriate memory reference commands to obtain the oper-

ands and generates the subroutine transfer into a program which will do the floating point arithmetic operation.

As another example, it may be that there are nested parenthetical expressions (in an arithmetic statement) requiring the use of several working addresses or several accumulators. The intermediate language is set up on the basis that there are infinitely many accumulators, so as the nesting takes place the computation may move from one accumulator to the next. If the object computer has only one accumulator, then it is necessary that the assembly program generates "store in working address" commands and corresponding commands which will assemble these results as the closing parentheses occur. Also the object computer may have none, a few, or hundreds of index registers. Consequently, if the source language problem deals with subscripted variables, then the development of index operations may be quite different in one object computer as compared to another.

On the other hand, there are certain operations which the assembly program must do which are very similar in all computers. For example, there are transfers of control or subroutine transfers which are to absolute locations which cannot be determined in advance. This is true of transfers into future locations (not yet assembled), in that the number of commands down to that future position may vary substantially from one computer to another. Hence, the assembly program must keep lists of so-called future addresses or develop transfer vectors which will take care of these connections. This operation may be done in essentially the same way in any of a number of different computers.

TRANSFORMING THE SYSTEM ONTO A NEW COMPUTER

If it is desired to transform the system onto a new computer, it is only necessary to rewrite the assembly program for the new computer. Once this assembly program has been completed, then the preprocessor and the translator may be processed on any computer for which the translation system exists, so as to obtain them in the intermediate language form. This intermediate language form may then be assembled on the new computer. Then one has a preprocessor, translator and assembly program which will run on the new computer and any problems stated in the appropriate source language may be processed on the new system.

Actually, the assembly program on the old computer may be modified so as to generate new computer commands on the old computer. In this way the "bookkeeping" portions of the assembly program need not be hand written for the new computer. The intermediate language version of the assembly routine may then be processed by the modified assembly program on the old computer. The output is an assembly program which runs on the new computer and transforms intermediate language statements into new computer code.

THE ANALYZER

In the statement of a problem in a language like ALGOL there is considerable redundancy, particularly between declaration-type statements and the way the corresponding variables are handled in the body of the program. This has led to the study of a proposed analysis system which will look at the ordinary statements of a program and will attempt to generate the appropriate declaration statements.

There are two schools of thought about the merits of doing this. One school says that the redundancy in the statement of the problem is of importance because this allows the translator to detect errors in a problem. The other school of thought says that, if a person is to move on to more and more complicated situations, it is necessary that the language that he uses be efficient.

In the current approach to this analysis problem, the idea is to analyze the statements of the problem and generate appropriate declaration statements whenever no ambiguity is involved. In cases where it is not quite clear what declaration should be made, that which would most frequently be correct will be made; the information will be printed out indicating to the operator that such a declaration has been made but that there might be some question about it being the proper one. In this way, an operator can program a large problem, run it through the analysis program, and get a list of questionable declarations which he can review for correctness. This analysis routine will also inspect multiply-subscripted variables and decide how to handle index registers. This allows the possibility of carrying values for a particular index appearing as a subscript in a number of different positions. For example, in a doubly-subscripted variable, a single index in the first position will mean one thing, whereas

the same index in the second position will mean a different thing. In one case, it is the element in the row; in the other case, it is the column in the array. If the analysis system detects such use of the index register, then it can issue index modification statements which will carry both values of the resulting index and for any incrementation increment each one in the appropriate way. In this way, in the current state of the computing machine art, more efficient object programs can be compiled than can be compiled by the more general conventional way of multiplying the indices in the various positions by the appropriate constants.

All the inspection of the source program for consistency and for obeying of syntactical rules is moved up into this analysis program. This permits the preprocessor translator and assembly programs to run at maximum speed.

THE TRANSLATOR

The translator inspects the syntactic aspects of the output of the preprocessor and transforms it into something more or less equivalent to a Polish string.

Thus, the various operators, separators and bracketing symbols are all ranked. These are processed from left to right and, depending upon rank, either intermediate language commands are developed in the output or entries are made in a stack. With each addition to the object program the rank of the last entry in the stack is inspected to determine whether processing continues with the source language or with the last entry in the stack.

The intermediate language has been designed in such a way that the commands for the output are all obtained from a table. This means that the translation process runs exceedingly fast. For example, speeds above 20,000 per minute can be expected from currently available commercial high-speed computers.

In fact, by proper preprocessing and analysis, translations may be combined with assembly.

A PRACTICAL SYSTEM

The conceptual division into analyzer, preprocessor, translator and assembler is convenient for exposition purposes, or for purposes of task assignment in a group effort. However,

since much of the action during assembly is still computer independent, another approach is suggested.

A basic subset of ALGOL is chosen including assignment statements not using multiply and divide operators nor parentheses. Procedures without recursion, with no arguments, are used. Only integral-type variables are used. Boolean quantities are represented by zero and non-zero integers. To this is added a declaration of the number of characters per word. A concatenation operator and a first-character operator are added to the system. Two undefined procedures named READ and WRITE are included. READ brings the next line of input (next card or characters from a typewriter down to a carriage return) and WRITE records on an output medium one line of characters.

Suppose this language is called L. A translator, T_L^A, for this language L is required on some computer, A. An important point is that this translator is also written in its basic language L (subset of ALGOL with extensions). All translators written in this basic language can be transformed into machine language for computer A by translator T_L^A.

In order to transform a body of coding in language L into programs on a new computer, B, a new translator, T_L^B, is required. This is accomplished by modifying the command generators of translator T_L^A so as to generate commands for computer B. The magnitude of this task is less than one expects due to the simplified basic language L, which is such that the command generation is a table-look-up process. Thus, by replacing the tables of T_L^A, a translator T_L^B is obtained which runs on computer A and generates commands for computer B.

The translator T_L^B is processed by translator T_A^A, obtaining a translator T_B^B in the machine language of computer B. This translator will generate programs for computer B.

In this transferring to computer B the procedures READ and WRITE must be hand written in the machine language of computer B.

SUMMARY

Two schemes have been described. The first is a three- or four-stage (conceptually) processor in which only the last stage is essentially computer dependent. In converting to a new computer, parts of the last stage (assembly program) must be hand modified. In the second scheme a subset of ALGOL with extensions is used to describe the translators and other processors. The translator for this simpler system must be hand modified for a new computer system. The system is sufficiently simple so that this modification consists of changing tables and input-output routines.

The following students at the University of California are working on various parts of the above schemes: Jim Spitze, Gary Anderson, Bill Keese, Ralph Middleditch, Ralph Love, Norman Josephson, Niklaus Wirth and Ivan Flores.