An introduction to compiler writing

By J. M. Forbes*

This paper is based on a talk given to the Birmingham branch of the B.C.S. in January 1965. It refers to problems met with in developing a translator and a compiler for LEO III computers. These have a single-address order code, one level of store is usually recommended, and the order code is designed to handle and store data directly in any radix; decimal, binary, or a mixed radix such as sterling.

The compilers referred to in the title of this article are two in number. Both are working compilers used every day by programmers on the LEO III range of computers, and it should be stressed that what is described in this article are features of compilers which have been incorporated and proved.

The two compilers mentioned are both scientific and commercial compilers, but it is their commercial aspects which are considered here; one is an intermediate language compiler and the other a full Autocode compiler.

The words *Compiler* and *Translator* are often used to mean very much the same thing, and will, for this article at least, both be defined in general terms to mean computer programs which turn a higher-level language into a lower-level language. This will enable both the Intercode translator and the CLEO compiler to be safely considered within the title.

It must not be thought that compiler writing is necessarily very different from other kinds of programming. Just as in other data-processing applications one starts with data and ends with results; the only difference is that the data in compilers happens to be a computer program. This computer program which is compiled is known as the *source program* and the program produced by the compiler is the *object program.*

Further, within compilers, the process of going from data to results may be divided into several parts. In commercial suites these parts tend to be separate programs; in compilers they tend to be passes of the same program because in this case it is known in advance that one process must immediately follow in time some previous process.

A first cursory glance at compilers will show that even the simplest must probably exist in multi-pass form. That is to say the data must be processed several times. This is illustrated by the problem of addressing: that is of allocating a final address by the compiler irrespective of whether it is a relocatable or an absolute address. Two main types of address exist: addresses of data and addresses of program, probably in the form of sequence-change addresses. If data are declared or described at the start of a program, that is to say the nature and length of the data are fed to the compiler before the instructions, then the address of any piece of data may be calculated and stored before any attempt is made to translate any instructions. When the instructions themselves are translated the address of any piece of data is immediately available. However, it may be that for other compelling reasons the data for a program lies in store in a position after the computer code instructions. Thus before the instructions can be translated, and their total length calculated, it is only possible to calculate the relative address of each piece of data.

In the case of sequence changes the problem is more acute. There are three reasons why the address can not immediately be determined. These occur when any kind of labelling system exists in the higher-level language, or the length of the object program instruction is variable, or if there is not a one-to-one correspondence between source language and the object program. In these cases it is not possible to know the final address of the destination of any sequence change, if that sequence change is to a point later in the program, until such time as the instruction, which is the destination of the sequence change has been reached in the translation process. There are three possible solutions to the problem. First the *load and go* method. In this method the object program is built up in the store during compilation and when compilation is finished the object program may be immediately run. The second method is a variation on the load and go method in which, after the compilation process has been finished, the object program is written onto some output medium for later refeeding. In both these cases it is possible to postpone the calculation of the final address until such time as the whole program has been through the basic compilation process. However, both these processes suffer from the drawback that space must be taken in the store to hold the object program as it is being built up, and further, in the case of higher-level languages, the translation process may be of such complexity that the object code to be produced is not immediately apparent. For this reason resort must be made to the third method—the *multi-pass* method, in which lists may be built up in one pass of the program and be used in a subsequent pass to help the translation process.

The two compilers to be described further are both multi-pass compilers and two features of the INTERCODE translator should be particularly noted.

* *English Electric–Leo–Marconi Computers Limited, Hartree House, Queensway, London, W.2.*

INTERCODE

It is established practice among LEO users that a data-processing system will not start to process any data until that data has been thoroughly vetted. This is equally true of compilers. If a compiler does not vet its data thoroughly it is likely to get into trouble itself or even worse produce wrong object program coding without any indication that this is the case. INTERCODE is an intermediate language with upwards of 150 actions with facilities for constants and tables and an expansion factor of about 1·0 to 1·5. Once the function part of any of these 150 actions has been recognized there remains the problem of vetting the other constituent parts of the action, of which there may be up to 5.

It is a further requirement of the INTERCODE system that the nature of errors detected be printed out with the instruction in the program listing. The system adopted is to divide errors into two classes—disastrous and others. Disastrous errors would stop the program running correctly and are indicated by 5 special marks in the program listing. In addition the offending entry is designated.

To carry out individual tests on each of these 150 actions, or even to identify them individually and to carry out tests by subroutine, is bound to be a fairly long process. Instead a table is held, whose entries are made up of an instruction number and certain coded details, each of which comprise a number of bits, which signify what the valid field values of the instruction are. This table is held in action-number order and, in fact, one entry is held only for the highest-numbered action in a group which share common checking characteristics. An additional complication is that certain actions may have continuation lines, and this swells the number of entries. However, the total amount of space used is equivalent only to 85 instructions.

The actual checking process consists of doing a "table look-up," finding the relevant checking constant and using that constant to enter a number of routines to carry out the appropriate detailed checking. This is not claimed to be some new and marvellous technique peculiar to compilers; there are many commercial data-vetting programs which use similar techniques, but what it does illustrate, is that in an area where it may be thought that particular compiler problems exist, the solution lies in an approach which could well be applied to programs in general.

Another example taken from the INTERCODE Translator does not point to the same conclusion.

The problems of sequence changes were previously hinted at. Here the INTERCODE requirements must first be explained. As well as producing an object code, the translator produces a print-out of the source language in which all sequence changes are marked. That is to say that against each instruction which is the destination of a sequence change, an indication is given of the source of the sequence change or changes. This, in itself, obviously requires two passes; in fact the INTERCODE Translator consists of three passes.

The most obvious way of solving this problem is to form up in the first pass a list of sequence changes and sort them into order of destination. In the second pass, as each destination is reached, the entry in the list is replaced by the calculated computer code address; sequence changes are then sorted back into order of source so that, in the final pass, destination addresses may be placed in sequence changes.

This system implies that a large expanse of store is required for sequence changes in each of these three passes. An improved system has therefore been introduced whereby sequence changes are divided into sequence changes forwards and sequence changes backwards. Sequence changes forwards are sequence changes to points later in the program. In the first pass only sequence changes backwards are noted, and sorted into destination order, giving an obvious space saving in Pass 1.

In Pass 2, the translation of these backward sequence changes can be completed, by calculating the final address when the destination is reached and, as this is known to be earlier in the program than the source, this final address can be used to complete the translation of the sequence change. A further advantage accrues in that as each source is reached, so this particular entry may be removed from this backward sequence change list.

The translation of forward sequence changes does not now start until Pass 2. Any forward sequence change is considered between its source and its destination. When the destination is reached the source may be printed and the final destination address associated with the source. At this point in time, the sequence change can be removed from the active list, and if this sequence change does not have as its destination the start of a procedure, that is to say its destination is expressed in the source language as being relative to a reference point, it is placed in a new list, which expands as Pass 2 progresses and is used to assign the final address in Pass 3.

This point raises a question of a more general nature. In any compilation process it seems that there are bound to be lists of theoretically indefinite length. If they are made too small, language restrictions and irate programmers result. If they are too large, they consume too much store space which the compiler writer can usually ill afford. It is up to the compiler writer to try to resolve this dilemma.

CLEO

The remaining examples are taken from CLEO. LEO Computers Ltd. started the development of CLEO, which is a full-scale Autocode, in 1961. It is now being used by the majority of those English Electric–LEO–Marconi Computers Ltd. customers who have a member of the LEO range of computers. This has been a most encouraging response; more than one user has said that all his future programming will be done in CLEO

and at the time of writing at least 25 of the first 30 users of LEO equipment have used CLEO to some extent.

Part of this success must be attributed to the efficiency of its compiler. An efficiency of 75% to 80% in mainframe running time and store usage has been achieved. These facts suggest that CLEO is the widest used Autocode in this country for commercial purposes.

CLEO, in common with other Autocodes, has two main divisions; the *procedure* division and the *data* division. These two divisions reflect the two main problems which occupy the attention of the CLEO compiler.

These two main problems are

(i) to break down the procedures into their constituent parts and into computer code;
(ii) to allocate addresses to identifiers and ensure these addresses are associated with the references to their identifiers in the procedures.

It should be added that the compilation process from CLEO to machine code is divided into two parts: from CLEO to INTERCODE and then from INTERCODE to machine code.

Perhaps the best way to see how the compiler deals with some of these problems is to take a simple example and follow this through from source language to computer code.

Consider the CLEO command

SET NETPAY = GROSSPAY − DEDUCTIONS

which appears in the procedure division and assumes that within the data description, the three identifiers NETPAY, GROSSPAY and DEDUCTIONS are declared.

The CLEO compiler is divided into six passes, one of which may be repeated several times. Two types of run are catered for: an initial run—the first compilation of any program, and an amendment run—for subsequent corrections.

The tasks of the first pass are straightforward and consist of producing an up-to-date listing of the source program on the printer and writing it onto magnetic tape. The program, as held on magnetic tape, may be considered in two parts; the data description and the procedure description. The next two passes process these separately.

First the data description is processed, and as a result of this addresses are allocated to identifiers and these are written onto an output tape. Thus, at this stage the three identifiers NETPAY, GROSSPAY and DEDUCTION appear on the output tape with an address allocated to each.

Secondly, the procedures are dealt with and the result of this is once again a list of identifiers, but with each identifier there is associated a tag which says how the identifier is used, e.g. as a multiplier in an arithmetic command, in a print command, etc.

The order of the identifiers is now one in which it would be possible to obey the program. It should be mentioned that LEO III is basically a one-address machine with an accumulator. Considering the given example, the three identifiers would have been output as: Select GROSSPAY, Subtract DEDUCTIONS and Transfer to NETPAY.

The scanning process

This conversion or scanning process forms an important and fairly complex part of the compiler, and it is worth while studying certain aspects of this problem in more detail.

This scanning process applies mainly to SET commands which are used to carry out all arithmetic processes in CLEO and which take the form

SET identifier = arithmetic expression.

Now the object program which has to be generated from SET NETPAY = GROSSPAY − DEDUCTIONS is perhaps, fairly obvious, but the CLEO language allows for rather more complex expressions than this, so that a procedure must be found for turning all expressions of whatever complexity into an accurate object program. The method employed is based on placing the arithmetic expressions into a Reverse Polish String (Łukasiewicz, 1921, 1929) and when certain conditions are satisfied, generating appropriate one-address coding.

Operators (that is to say +, −, =, etc.) and operands (identifiers and literals) are placed in a stack (a list controlled by a modifier) which works on the 'last in first out' principle. Each operator is assigned a priority code number as follows:

+, −	30
×, ÷	40
=	20
Start expression	10
End expression	10

Operators, in fact, have their own stack. They are placed on this stack only if the code number is greater than that of the operator on the top of the operator stack. If not, a generating routine is entered.

The generating routine processes the top operand and the top operator to produce an output item. These are then removed from their respective stacks and the operator under consideration is compared with the new operator which is on the top of the stack, and the process continues, with operators either being placed on top of the stack, or generated and output until such time as "end of expression" is recognized.

Perhaps an example will serve to illustrate how this works in practice.

Consider SET $A = B + C \times D$

The obvious problem here is to avoid performing the addition before the multiplication.

The operators in this command are start expression (code value 10), = (code value 20), + (code value 30), × (code value 40), and "end of expression" (code value 10). It is therefore not until the end of the expression is recognized that any generation takes place, and as the operand stack at this time holds, from the top, *DCBA*, the output that is produced from this expression is

Select *D* as multiplicand, multiply by *C*, add *B*, transfer to *A*.

To show that output may be generated before the end of expression, consider the example

SET $A = C \times D + B$.

In this case the operators at the time when the plus is being dealt with are start expression (10), = (20), and × (40). As the next operator value is 30 generation can take place. This generates "multiply *C* by *D*" and then allows + to be placed in the operator stack as before.

A particular exception is caused by brackets. An open bracket is always put on the operator stack without any comparison and, when the corresponding closing bracket is found, generation must take place until the open bracket comes to the top of the operator stack, from where it is then removed.

A similar scanning process is used to deal with conditions or decision-making commands. The scanner is able to discern whether it is in a SET or a conditional command, and deal with "equals" accordingly.

At this stage in the compilation process, when the scanning has been completed, two magnetic-tape files are in existence. One, it will be remembered, contains a list of identifiers, built up from the data description and containing such details as allocated address, size and radix of each identifier. The other, produced from the procedures, also contains a list of identifiers, but in this case attached to each identifier is a tag which signifies how and in what kind of command each identifier is used.

In the case of the original example the output produced by the scanner would be

Select GROSSPAY, Subtract TAX, Transfer to NETPAY

Additionally, with each of the three identifiers a tag is attached which indicates that they occurred in SET commands.

The remaining passes

The problem now is to merge these two files in such a way that either the translation process is completed or sufficient information of the right sort is brought together in the right form to enable the translation process to be completed in a subsequent pass.

The merging process which is performed by the next pass produces as its output one file, on which each identifier has had its allocated address associated with it. A large expanse of store is filled with data description, and the file produced from the procedures is passed through and a match sought between the data description in the store and the identifiers occurring in the procedures. Where a match is found the data description appropriate to the command in which the identifier occurred is associated with the identifier and written on the output file. If the data description is so large that it cannot all be held in store at one time the process is repeated as many times as are necessary to complete this merging process. At the end of this pass the identifiers GROSSPAY, TAX, NETPAY retain their original position on the procedures file, but now have associated with them their allocated address and declared radix (e.g. sterling, decimal, etc.).

In the final passes of the compiler two tasks remain outstanding. These are to complete the translation of those commands (SET and conditional commands) whose breakdown began in the procedure editing pass, and to translate those commands which it has been impossible to complete since part of the data description pertaining to them was not previously available.

First the completion of the translation of the commands dealt with by the scanner: these commands, it will be recalled, had to be considered without any reference to their data description. The length and radix of all items are known. It is now apparent that a programmer has specified, for example, that he wants to add a sterling number to a decimal number and produce a binary result. This is a perfectly possible thing to do in theory, although some conversions have to be introduced into the object-program coding; or that the interval between successive occurrences of any item is greater than unity and that some kind of arithmetic or shifting must be performed on any subscript before it is used to carry out the modification. The original command containing the identifiers GROSSPAY, TAX, NETPAY would be rescanned with this in mind.

But this part of the translation process is by no means all loss. With all the information required now available a second look can be taken at many of these commands and, in the light of this complete information now available, a fair degree of optimization can now take place.

For example, the occurrence of repeated subscripts can be recognized or advantage taken of the state of accumulators at a given point in the program.

The translation of the other commands, which tend in practice to be the more powerful commands, is also a major task for the later passes of the compiler. The translation of a command such as MOVE *A* to *B*, while it may be a simple process, can also be one of considerable complexity. Thus a MOVE command can apply to any level of data ranging from the simplest single item to a large record with complex layout which may not be completely identical in source and destination. Obviously, in cases such as this, rules have to be laid down on how the commands are translated. For example, the compiler writer must decide if and where there is a breakeven point between selecting and transferring between source and destination, and bulk moving

with its larger initial overheads but quicker operation once the initial set-up has been carried out.

There are further cases of this necessity to break down records or groups into their constituent parts in the input and output commands such as PRINT and FILE. The translation of these commands, do, of course, make use of the standard INTERCODE input/output facilities, enjoyed by all LEO users. Despite this assistance, however, the formation of the relevant object coding to deal with such commands is relatively speaking a matter of hard work and much coding in the latter passes.

At the end of the compiling process an INTERCODE object program has been produced, and from then on the process of reaching computer code is as for INTERCODE.

Conclusion

The case for automatic programming is well known; the two main disadvantages, inefficiency in the object program and the length of time it takes to compile, are probably equally well known. Unfortunately, these two disadvantages tend to pull in opposite directions. If the compilation process took longer one could have a more efficient object program.

Further, compilers usually have to be written for minimal configurations. This tends to reduce the amount of store available to the compiler writer and hence the number of instructions, and this means that the number of passes must be increased.

The inefficiency factor arises partly from the need to deal with situations in a general way. For example, at the entry to any routine one cannot make any assumptions about what is in any of the accumulators or modifiers or what radix is set. Therefore the compiler writer has to take precautions and perhaps insert some extra instructions, in case the correct values are not in the relevant registers. Any hand coder would probably only do this where necessary.

Furthermore a compiler can make no assumptions about the likelihood or otherwise of any particular routine of a program being obeyed. To a compiler they are all one. And it is probably because of this that compilers for machines with two levels of storage have tended to be less efficient than others.

References

ŁUKASIEWICZ, J. (1921). "Logika dwuwartościowa" (Two-valued logic), *Przegląd Filozoficzny*, Vol. 23, p. 189.

ŁUKASIEWICZ, J. (1929). *Elementy logiki matematyczny* (Elements of mathematical logic), Warsaw.

THOMPSON, T. R. (1962). "Fundamental principles of expressing a procedure for a computer application," *The Computer Journal*, Vol. 5, p. 164.

Book Review

Excitation Control, by G. M. Ulanov, 1964; 100 pages. (Oxford: *Pergamon Press Ltd.*, 30s.) (International Series of Monographs on Electronics and Instrumentation, Volume 29.)

The *Pergamon Press* has established a high reputation for its valuable work in publishing translations of Russian technological works, especially in the field of control. It will lose that reputation if it publishes more translations as bad as that of *Excitation Control*. The book has more errors in it than a colander has holes. They abound in the text, in the mathematical equations and in the diagrams; on one page alone there are seven errors. The post-translation editor says: "In the main the author's terminology has been retained except in the cases where some ambiguity of ideas occurred, but it is hoped that in this, the edited versions [*sic*] of the translation, any errors and imperfections have been reduced to a minimum." A post-translation editor should, above all, be technically knowledgeable in the field covered by the book; it is hard to believe this of one who can, to mention a few examples, print "feeding voltage" for "input voltage," "extreme" for "extremum," "transfer" for "transient" (many times), "multiplier" for "factor," "hydroscopic" for "hydraulic," and "pressing device" for "screw-down" of a rolling mill. And what can one think of editing which allows the German mathematician Weierstrass to appear, after a double transliteration, as Veiershtrass?

With all these errors it becomes really hard work to find out what the book is all about, and the title certainly does not help the English reader. In the broad, it concerns the application of the principle of invariance, or the use of feed-forward, open-loop control paths to compensate for disturbances, including load variations (when they can be measured), and to make for improved following of input signals. Chapter I is a brief historical survey, mostly of developments in the Soviet Union, Chapter II a series of descriptions of control systems using the method, Chapter III a short run-down of the theory, and Chapter IV examples of calculations. The major part of the work, in Chapter II, consists of a brief précis of each of a number of published works describing control systems. These are mostly much too brief for the reader to get a good idea even of the broad outlines. There is, however, a full list of references, 42 out of 43 of them to Russian publications.

The book would be of use to the reader whose prime interest is in studying the state of the art in the Soviet Union, to whom the survey and bibliography of published work would be of value. He would probably find it worth while to consult the original, and it would be interesting to know how many of the errors in diagrams and in equations are to be found there. The book is not recommended to those whose interests in the field are purely technological. Perhaps the reference to itself, on two occasions, as a "brochure" is not, after all, one of the mis-translations.

R. H. TIZARD