WHETSONE ALGOL REVISITED

OR

CONFESSI0NS OF A COMPILER WRITER

by

B. RANDELL

The English Electric Co. Limited, Whetstone, Leics.

Paper presented at the first Compiler writing Seminar, 20th May, 1964,

Brighton College of Technology.

INTRODUCTION

A complete and detailed account of the Whetstone KDF9 Algol Compiler has been given in the book 'Algol 60 Implementation' (4). The present paper uses the wisdom born of hindsight in an attempt at a critical re-appraisal of the Whetstone Compiler and the implementation techniques which are used in it.

DESIGN AIMS

The Whetstone Compiler, which is designed for use during the debugging of Algol programs, consists of a Translator and a Control Routine. The Translator is a single-pass translator producing an object program which is interpreted at run time by the Control Routine. After an Algol program has been debugged it can be translated using the Kidsgrove KDF9 Algol Compiler, a multi-pass optimising compiler which produces a machine code object program. The decision to develop an 'Algol System' incorporating two separate Algol compilers was based on the fact that two of the most desirable qualities of a compiler, namely speed of translation, and efficiency of object program produced, are to a large extent incompatible. Thus in the Whetstone Compiler efficiency of object program is sacrificed in order to obtain fast translation together with extensive program checking and testing facilities, whilst in the Kidsgrove Compiler considerable effort is put into transforming the Algol program into a form which will make the best possible use of the computer hardware.

This 'twin compiler' technique, though in many ways very convenient, does have certain disadvantages. Since the characteristics of the two compilers are so distinct, a programmer must decide carefully when to assume that his program is virtually error-free and worth compiling using the optimising compiler. Furthermore the system is not very suitable for large programs which are continually being refined and developed.

The most obvious alternative to the twin compiler system is the compiler which is organised in such a way that the sections which attempt to optimise the object program are bypassed unless the user specifies otherwise. Superficially such a compiler would be just as satisfactory as two separate compilers, with the additional advantage of making the change over from unoptimised to optimised translation extremely easy. However, as will be seen from an examination of the Whetstone and the Kidsgrove compilers, fast translation and optimised translation can use widely differing techniques which may be difficult to combine into a single compiler.

A near-ideal alternative would be an efficient compiler which could incorporate minor changes to a source program with a minimum of retranslation and which includes effective program checking and testing facilities. (A possible method of implementing a compiler which permits such 'incremental translation' has been described by Evans, Perlis and Van Zoeren (2).) The lack of this ability to perform 'incremental translation' can be compensated for by the manual splitting up of a program into 'segments' which are then translated separately. When changes are made to a program only the segments which have been affected have to be translated. In KDF9 Algol segments must be 'complete' procedures (i.e. in which all identifiers are either formal parameters or local to the procedure), and are also used to indicate program storage overlays. However, because the two KDF9 Algol compilers produce such different styles of object program, all segments of an Algol program must be processed by the same compiler - this is unfortunate as the facility of combining segments produced by either compiler would be very useful for large-scale programs which are gradually modified and developed over a long period of time.

The original premise in this discussion of the twin compiler system was that fast translation was incompatible with the production of efficient object programs. This is surely true at the present time, since the production of an efficient object program needs a sophisticated method of scanning a source program in order to detect restricted uses of certain language forms which are then treated specially so as to make the best possible use of all available machine facilities. It is to be hoped that in the future the design of programming languages and machines will become more systematic and elegant so that, because of the lack of awkward restrictions and special cases, a more straightforward process will suffice to transform a source program into an efficient object program.

SINGLE-PASS TRANSLATION

One of the principal characteristics of the Whetstone Compiler is that it performs the transformation of the Algol source program as the source program is being read, so that the object program is complete and ready to be obeyed almost immediately after the end of the Algol text has been reached. This so-called 'single-pass' method of translation has obvious advantages during program testing, since the programmer can virtually ignore the existence of the compiler, because his program, albeit in Algol, is read into the computer at normal paper tape speed. These advantages are however largely nullified when

time-sharing machines, or a system for keeping programs which are currently under test in some form of backing storage, are considered.

Single-pass translation, besides being a rather pleasing concept, is also an interesting challenge to the compiler writer. The problem is that the generation of object program coding must proceed virtually concurrently with the analysis by the compiler of the language instructions forming the source program - this analysis can be based only on the current portion of source program text, together with any information which has been gleaned from text which has already been scanned. There is no possibility of a preliminary scan to collect and classify the various identifiers used in a program, leave alone the more sophisticated forms of source program analysis which are attempted in many multi-pass compilers.

The need to generate object program as the source program is analysed would obviously be greatly facilitated if each section of source program could be correctly and completely analysed using only information that has already been scanned. This is unfortunately not the case with Algol, so the Whetstone Compiler uses where necessary a technique of generating 'incomplete' object program operations which are completed when the missing information becomes available. This technique involves designing the object program so that the various operations which would be produced from a temporarily ambiguous source language construction have a basic similarity - this was possible in the Whetstone Compiler where the object program was to be interpreted, and was therefore free of the constraints imposed by machine hardware, such as the need to give explicit instructions for floating integers.

Example

```
begin       real procedure p;
  begin          ...
                 p = a + b;
                 ...
  end p ;
  integer a; real b;
```

The declarations of the identifiers a and b which are used as non-local variables inside procedure p can, as above, be given after the declaration of p. Thus during the translation of the statement 'p := a + b' it is not known whether a and b are real or integer variables or even real or integer function designators without parameters. As a result the object program operations produced by the Whetstone Compiler corresponding to identifiers which are simple variables or parameter less function

designators take the same basic format, in fact an 8-bit operation code, followed by a 16-bit address field.

The necessity of designing the object program so that certain classes of operations have the same basic format would be removed if each operation carried with it the address of its successor (an order code of this form was of course common with early computers where 'optimum coding' was used to minimise drum latency time, e.g. the IBM650). With such an order code the object program is effectively stored as a 'list structure' and operations of varying size could easily be added to or deleted from any part of the program at any stage during translation. However an object program of this form would be considerably larger than the equivalent serial object program because of all the extra program addresses.

The technique used in the Whetstone Compiler to enable incomplete operations to be replaced by complete operations when the missing information becomes available is dependent on having the whole of the partially generated object program available in core storage. The address fields of sequences of incomplete operations corresponding to each identifier for which no declaration information is available are 'chained' together, and the chains are traced through when the operations and their address fields are being completed. Thus although the Whetstone Compiler uses only a single scan through the Algol source program, it does this at the cost of a set of what could be called small 'scans' through short sequences of object program operations.

One of the most obvious ways of avoiding either some such technique of chaining, or a preliminary scan of the Algol text just to collect declaration information, is to impose the restriction that the declarations of all identifiers other than labels must precede their use. (No ambiguity results if only a single category of identifiers can be used before being declared - labels are the obvious choice as the 'single category' because the translation of forward jumps will in any case need some such technique as chaining). A restriction of this form is normally a source of annoyance only to programmers who have had access to a compiler which does not need this restriction, though it does preclude the writing of explicit mutually recursive procedures. However a pair of procedures can be made to call each other recursively without infringing the rule that identifiers must be declared before being used, by giving one procedure as a parameter to the other procedure.

Example

```
begin          procedure p ;  begin     ... ; q ; ... end ;
          procedure q ;  begin     ... ; p ; ... end ;
```

can be rewritten as

```
begin          procedure p(Q) ; procedure Q; begin ... ; Q ; ... end ;
```

```
procedure q    ; begin ... ; p(q) ; ... end ;
```

which has exactly the same effect, but avoids using the identifier q before its declaration.

The technique of chaining used in the Whetstone Compiler to permit single-pass translation of unrestricted Algol involves use of a 'name list'. The name list is effectively a pushdown store containing items which give an identifier and either its declaration details or the addresses of the start and finish of the chain of incomplete operations which has been generated because declaration information was not available. In a large program the name list can take a considerable amount of storage, because at any given point during translation the name list contains a number of groups of items, corresponding to the current block and all its surrounding blocks, where each group contains one three-word item for each different identifier that has been declared or used in the corresponding block. In fact compared to the name list, the translator stack, which is used to re-order arithmetic operations, and to analyse statement structure, normally takes very little space. Thus on a KDF9 with a limited amount of core storage, the amount of space required for the name list is one of the main limiting factors on the size of Algol program that can be translated. An obvious improvement that could be made to the compiler would be for it to use backing storage for parts of the name list not in current use; this could be arranged quite easily because the name list is organised effectively as a push down store, and only the items corresponding to the current block, and occasionally the immediately surrounding block, are ever needed at any one time.

It is because the name list contains entries corresponding to declared identifiers which have been used but not declared in a block that only the top one or two groups of items ever have to be searched to find whether there is an entry corresponding to a given identifier. Thus the time taken to construct and later to follow through the chains of incomplete operations is at least partly offset by the fact that very little time is spent searching identifier lists. However at first sight it seems unnecessary for the name list to contain an item for each block which a given identifier is used.

Example

```
begin

        real a, b;

        a:= 0 ;

        begin real c ;

            c:= a ;

            b:= 0 ;
```

```
                ...
```

After the translator has processed the statement 'b := 0' two groups of items will have been set up in the name list. The first group, corresponding to the outer block, will consist of two entries, giving declaration information for the identifiers a and b; the second group, corresponding to the inner block, will consist of an entry giving declaration information for c, and two entries giving addresses of the chains of incomplete operations generated from the uses of identifiers a and b.

The declaration of an identifier can follow its use only if it is a label, or it used within a procedure body as a non-local variable or within a switch declaration (it is convenient to think of a switch as being a procedure of type label). Thus in the above example within the inner block it is obvious, from the way which they are used, that a and b are not labels, and hence must correspond to the a and b declared to be real variables in the outer block. Hence it would appear worththwhile to avoid generating name list entries, corresponding to identifiers whose uses proves that they cannot be labels, if the name list groups for any containing blocks (as opposed to procedures) contain the same identifiers. However these apparently superfluous name list entries also serve as a record that an identifier has been used as, say, a variable in a given block, and hence cannot so be given as a label. Without these name list entries, or some equivalent means storing this information, it would not be possible for the translator to check out this type of error.

Example

```
     begin     real a ;

               a := 0 ;

               begin          real b, c ;

                         a := b ;

                    a:   b := b + 1 ;

                         ...
```

The identifier a in the statement 'a := b' would be taken as referring to the real variable a declared in the outer block. However, at the point where a is given as a label an entry would be added to the group of name list entries corresponding to the inner block, and the inconsistent use being made of a would be detected.

A possible compromise would be for the translator to search the groups of name list entries for the surrounding blocks until a procedure block is reached, and to generate chains of incomplete operations only for those

identifiers for which no name list entries are found, but to still add an entry to the name list for each identifier which is used, so that inconsistent uses of identifiers are still detected. Thus one could save dome chaining at the cost of some extra name list searching.

OBJECT PROGRAM INTERPRETATION

The second major characteristic of the Whetstone Compiler is that the object program is not in machine code, but is obeyed interpretively by a 'Control Routine'. The most important reason for this is that it permits a complete check to be made of all the actions of the object program and all communication between program and programmer to be given in terms of the original Algol source program. Thus the result of each arithmetic operation is checked to be within the permitted range for integers or for real numbers, as appropriate; and the address given by each subscripted variable is checked to be within the storage area allocated to the corresponding array. Obviously the increased running time caused by interpretation can only be justified while a program is being checked out, and perhaps in some cases only during the initial stages of checking out.

Since the object program is being interpreted it is possible to allow the Control Routine to perform at run time some functions, such as deciding when real/integer conversions are necessary, which the single pass translation technique made inconvenient to perform at translation time. Thus the Control Routine completes the checking of the consistency of use of real, integer and Boolean variables, and of formal parameters with their corresponding actual parameters, and simulates automatic stack hardware working in the core store.

The first function of the Control Routine, the checking of all arithmetic operations, would well be performed be suitable checking and interruption hardware, linked to a protected 'director' or 'master control program', thus allowing the object program to be given in machine code. The director would use tables produced by the compiler so that any communication with the programmer could still be in terms of the original Algol source program.

The second $unction of the Control Routine, the checking of the consistency of use of variables of the various types, and the related question of deciding when real/integer conversions are necessary, is not performed by the Translator because the necessary declaration information is not usually available so that incomplete object program operations have to be generated and completed later. In fact the Control Routine stores the intermediate results of arithmetic expressions as double-word 'accumulators', one word containing the numerical value of the intermediate result, the other a bit pattern indicating its type; the action of each arithmetic operation is dependent on the bit patterns given in the accumulators containing its operands.

The third function of the Control Routine is to organise the core storage as a stack for the storage of variables and arrays, and for the

calculation of expressions. As a result the object program representation of arithmetic expressions can be given in a form of Reverse Polish, and the Translator does not have to assign addresses to the intermediate results of expressions. This makes the object program very compact but means that the hardware facilities, such as the nesting store accumulator, provided on the KDF9 computer are virtually ignored. A description of the Kidsgrove which makes full use of all these facilities, has been given by Hawkins and Huxtable (3). However, it is possible that a compromise could be found between the facilities demanded by the object program (here provided by means a Control Routine) and the facilities which would be provided by machine hardware, which would enable a reasonably simple and elegant translator to be used to produce machine code object program which would run with an acceptable level of efficiency.

TRANSLATION TECHNIQUES

The Whetstone Compiler uses a development of the stack and priority level translation technique first described by Dijkstra (1) to analyse the Algo1 source program and to control the generation of the corresponding object program. The Translator is thus largely composed of a set of separate routines, corresponding to the various delimiters, which use the translator stack and a set of 'state variables', in which information about earlier sections of the source program is recorded.

The system of translation, though perhaps a little ad hoc, particularly in choice of state variables, has proved most satisfactory in practice, particularly regards speed of translation; hence the author's slight preference for a system of translation using syntactic routines (i.e. routines corresponding to the various syntactic units of Algol, which can call each other recursively) is based solely on the feeling that such a system is more elegant and is more directly related to the formal definition of the Algol language. In fact if an Algol compiler was available, perhaps on another computer, for use during the development of a translator there would be a strong incentive to base the translator on the syntactic routine system, and to benefit from the ease with which recursive routines could be written.

As has been mentioned earlier, the Whetstone Compiler, because it is performing a single pass translation on unrestricted Algol, does not have available the necessary declaration information which is required to analyse, at translation time, the types of the partial results that will be produced at run time during the evaluation of expressions. In fact if the declaration information was available then it would in general be fairly easy for the translator to perform all the necessary analysis and checking of types, and to generate the appropriate real/integer conversion instructions (see, for example, Randell and Russell (5), which also describes a simple method of computing at translation time any operations whose operands are numerical constants). However certain minor features of Algol are not immediately amenable to the use of translation-time type assessment. The first problem is that of the type of conditional expressions.

Example

```
x: = (if i=O then x else i) X (if j=0 then y else j) ;
```

If x and y are real variables and i and j are integers then, depending on the values of i and j, which are of course not known at translation time, the multiplication can be of two real numbers, two integers, or one real number and one integer. A reasonable solution to this problem is to define the type of a conditional expression to be real if any of the constituent simple expressions are of type real. The second problem, to which there is apparently no satisfactory unrestrictive solution, concerns the exponentiation operation.

Example

```
i ↑ j
```

where i and j are integers, is of type real if j is negative, but otherwise is of type integer.

As it is, at translation time the Whetstone Compiler does a partial check on the correctness of the types of identifiers used in expressions, by using the rules pertaining to a given operator to check, where possible, the types of identifiers which appear adjacent to the operator as its operands. No attempt is made to check that the types of the partial results that will be produced during the expression are compatible, and no use is made of any declaration information known about an identifier should it be available. As information about an identifier for which no declaration information is available is gradually accumulated from the various ways in which the identifier is used, a bit pattern is constructed in the corresponding name list entry. When the declaration information becomes available it is checked that it is compatible with the information given by the bit pattern. The bit pattern is formed from two constituent bit patterns, one indicating whether the identifier has been followed by an opening parenthesis or square bracket, the other giving type information. It has since been realised that because the list entry also contains an entry which indicates the number of dimensions if it is an array (or parameters if a procedure), and which is set to zero for scalars, the use of the first part of the bit pattern is an unnecessary complication.

The naive way in which for statements and subscripted variables are treated in the Whetstone Compiler could not be tolerated if it was not designed to be part a twin compiler system. No attempt is made to recognise simple uses of for statements or subscripted variables; instead the Control Routine always allows for the possibility that the full generality permitted in for statements and subscripted variables is being used. By way of contrast, in the Kidsgrove Optimising Compiler (see Hawkins and Huxtable (3)) considerable effort is expended on finding cases of restricted uses of for statements and subscripted variables, which are then translated into a form which makes efficient use of the KDF9's powerful indexing and loop counting facilities.

CONCLUSIONS

The design of the Whetstone Compiler has been greatly influenced by the fact that it is to be part of a twin compiler system. However the basic design could be used without too many major modifications to produce a fairly fast compiler which would generate an object program more nearly comparable with ordinary machine code. On the other hand the problem of efficient implementation of for statements and subscripted variables requires a more major effort, and is very dependent on the quality of the machine's indexing facilities and order code.

ACKNOWLEDGEMENTS

Mr. L. J. Russell, who has played a major part in the development and implementation of the Whetstone Compiler, has also assisted the author in the preparation of this paper, which is published by permission of the English Electric Company Limited.

REFERENCES

1. Dijkstra, E.W. (1961). Making a Translator for Algol 60. A.P.I.C. Bull. 7, pp. 3-11.

2. Evans, A., Perlis, A.J. and van Zoeren, H. (1961). The Use of Threaded Lists in Constructing a Combined Algol and Machine-Like Assembly Processor. Comm. A.C.M. 4, 1, pp. 36-41.

3. Hawkins, E.N. and Huxtable, D.H.R. (1963). A Multi-Pass Translation Scheme for Algol 60. "Annual Review in Automatic' Programming", vol. 3, pp.163-205. Pergamon Press, Oxford.

4. Randell, B. and Russell, 1.J. (1964). "Algol 60 Implementation". Academic Press, London.

5. Randell, B. and Russell, L.J. (1964). Single-Scan Techniques for the Translation of Arithmetic Expre3sions in Algol 60. J.A.C.M. 11, 2, pp. 159-167.