BIT 3 (1963), 145–166

THE DESIGN OF THE GIER ALGOL COMPILER PART II

PETER NAUR

Contents.

4. Multipass translation.

4.1. *Storage allocation during translation.*

The importance of the scheme used for storage allocation during translation forced itself upon our attention through our work with the DASK ALGOL compiler. This compiler was based on principles which ignored the problems attendant on using a machine with a non-homogeneous store. The ill effects of this approach showed themselves in two ways. First, the writing of the translator became a continual struggle (and a very time-consuming one) to fit the translator programs and tables into the machine. Second, the frequent transfers of programs

and tables to and from the drum limit the speed of the compiler to a level which is entirely out of harmony with the internal speed of the machine.

4.2. *The translation problem in machines with a non-homogeneous store.*

The translation of an ALGOL program consists basically in the combination of two bodies of information, viz. the information supplied by the ALGOL source program and that of the translation program (the translator). In a machine with a non-homogeneous store we will at any time during translation have only a small fraction of each of these two bodies at our disposal in the fast store. The principal aim in designing the storage allocation scheme is to avoid that the part of the bodies being in the focus changes rapidly in a random manner.

Part of this aim may be achieved by using sequential scanning of the program text. By this we will understand that each basic translation process treats all or nearly all the symbols of the text by taking them in order from one end of the program to the other, producing at the same time a transformed text as output. This method of processing obviously avoids random references to the source program.

Where the translator program is concerned the answer to the basic storage allocation requirement is to divide the translation process into a number of separate passes. In each pass we will then need only a fraction of the total translator program.

This still leaves to decide, what is the best number of translator passes? The conflicting factors are, on one hand, that if we use few passes then the translator program of each pass will be long and at a certain point we are again back in the situation where we have to divide the translator program between the fast and the backing store, and on the other that if we use a large number of passes, then the total cost of transferring the program from the backing store to the fast store once for each pass will become comparable to the time of making a very few references to each part of the program, but in a random order.

The actual proportions will depend strictly on the data transfer characteristics of the machine. In GIER ALGOL the drum is the only backing store available. If, as a first very rough guide, we consider the times required to refer to a word placed on the drum we get:

	Access time	Drum time	Total time
Random	0.05 mS	20 mS	20 mS
Sequential	0.05 -	0.5 -	0.55 -

Thus under these circumstances we may perform more than 35 sequential references to all words of a text during the time it takes to refer once to each word of the same text in a random manner.

If on the other hand we try to make a guess as to the number of passes which would be necessary in order to avoid drum transfers of translator program during each pass we can proceed as follows: The total size of the translator ought not to exceed 5000 words (DASK ALGOL has about 5500 40-bit words). If we remember that each pass will require space for tables the average size of the program of a pass ought to be about 500 words. This then indicates that about 10 translator passes might be expected.

Comparing these preliminary estimates we arrive at the following conclusion: The translator should be designed on the basis of the principle that each pass, including all necessary tables, must be small enough to be held in the core store. The exact number of passes, estimated to be about 10, should be adjusted to fit the most logical division of the translation process consistent with this principle and need not for its own sake be kept small.

4.3. *Intermediate languages.*

In a multiscan translator the choice of intermediate languages becomes of prime importance. The following conflicting factors have to be considered: From the point of view of the individual scans (or translator "passes") great flexibility and a great multitude of structures of varying length is desirable. From the point of view of the programs which have to perform the packing and unpacking of information, which is necessary during each scan, uniformity of structure is desirable. Uniformity is also desirable for translator checking purposes.

In view of these considerations the following compromise was adopted: All intermediate languages are expressed in terms of basic units of information each of 10 bits. This unit is called a "byte". Thus the input to each pass and the output from it will always consist of a uniform string of bytes. Within each intermediate language any amount of structure within this byte string may be employed in order to communicate more intricate structures. Thus, for example, a number in the original ALGOL program will in most of the intermediate languages be represented by 5 consecutive bytes, the first being a distinct "number mark" while the remaining four supply the actual value of the number. However, from the point of view of the general packing and unpacking program the various intermediate languages are undistinguishable, since they all consist of just a uniform byte string.

By this method the programs which will perform the packing and unpacking of the intermediate versions of the program and which will take them from the drum and store them on the drum will be the same for all passes. In addition since the output from each pass is always of the same form the same test output program can be used for all passes and can be coupled to the common pass administration.

4.4. *Reverse scans.*

In a multipass translator it is highly advantageous to let some of the scans be reverse scans, i.e. scans which start at the end of the program and move towards the begin. First of all the problem of forward references clearly is solved completely in this manner. In addition it becomes easy to eliminate syntactically incorrect sections of the program during the translation. This is important because it is highly desirable that an error which has been detected does not prevent the translation process from continuing to check the rest of the program. Two of the 9 passes of GIER ALGOL are reverse scans.

4.5. *The general pass administration.*

According to the description given above all passes use the same general pass administration. As far as the individual pass programs are concerned the general pass administration is a subroutine with two entries, one for input and one for output. In fact these two entries are almost completely independent since they use two independent buffers in their communication with the drum. The general pass administration uses 4 sections of 40 words each in the core store. Two of these are used as buffers for the input to the pass programs, the other two for the output from the pass program. Normally one of the input buffers holds the bytes of that section of the program which is presently being processed by the pass program while the other input buffer holds the next input drum track. When all the bytes on the active buffer have been used the transfer from drum of the next following drum track is initiated. At the same time the pass program can proceed to process the input bytes waiting in the other input buffer. A similar buffering technique is used on the output side.

The bytes are packed into the GIER-words with 4 bytes in a word. Thus one drum track holds 160 bytes. The unpacking (on the input side) and packing (on the output side) of the bytes into the words are performed by the general pass administration. This turns out to be a more time consuming process than the corresponding drum transfers. In fact

the average unpacking time per byte is about 220 microseconds, which means that the unpacking of the 160 bytes on a track will take about 35 milliseconds. This again means that except for collisions between drum transfers called from the input and output side the time for drum transfers will be negligible, owing to the parallel operation during drum transfers in GIER. The time for packing on the output side being about 260 microseconds per byte, the total time for the administration of a pass becomes about 77 milliseconds per drum track plus the time wasted due to collisions of drum transfers, about 5 milliseconds on the average. The processing time per byte of course varies greatly, the minimum being about 80 microseconds (direct copying from input to output). At an early stage of the development we guessed that an average byte processing time of 500 microseconds, corresponding to about 10 instruction execution times, might be expected. On this assumption the pass administration and processing times are comparable and the total time for 10 passes will be about 2 seconds per drum track. This proved to correspond closely to the final performance.

On the drum the tracks holding the partially translated program are used in a cyclic manner. About half the drum is available for this, while the rest of the drum holds the translator and the running system programs. Suppose that the available tracks are numbered from 1 to N. The first pass will then place its output in tracks no. 1 to M, say, where clearly $M \leqq N$. The second pass will take its input from tracks no. 1, 2, etc. and will place its output in tracks $M+1$, $M+2$, etc. When output to track no. N has been made the administration will continue to output into track no. 1, 2, etc. which presumably have now been released by the input side administration. This process is continued smoothly from one pass to the next, except for the case that the direction of scanning is reversed. This does not cause any difficulty however, since the cyclic use of the drum tracks may with equal ease take place in either direction. Only in the programs which perform the packing and unpacking a few changes are necessary.

4.6. *Distribution of tasks among the translation passes.*

Once the great advantages of a multipass compiler are established, the problem remains to distribute the translation tasks among the passes. As will be shown below the solution of this problem follows in a straightforward manner from a consideration of the basic characteristics of ALGOL 60, of our basic demands on the compiler, and of the multipass scheme itself. These characteristics are:

1. ALGOL 60 characteristics: identifiers of arbitrary length; the characteristics of quantities are given by explicit declarations which may appear scattered among the program text (labels, procedure bodies referring to each other).

2. Demand on the compiler: it should detect virtually any number of formal errors of a program in a single translation. No kind of error should cause the compiler to be thrown off the track.

3. Multipass scheme: the program, tables, and working space of each pass must be accomodated entirely within the core store.

The proper sequencing of passes may now be established as follows: First, it follows from points 1 and 2 that at least two major checking processes must be distinguished. The first must analyze and check the delimiter structure, without regard to the types and kinds of identifiers. When this has been completed another process will be able to check and analyze the operand structure. In addition to these two processes we must have a first pass which will convert the hardware representation used for input into a form which gives the ALGOL 60 symbols directly. As our first rough picture we may therefore distinguish the following 3 phases, numbered in anticipation of the following discussion:

Pass 1. Analysis of hardware representation (microscopic context).
Pass 3. Analysis of delimiter structure (intermediate level context).
Pass 6. Analysis of types and kinds of quantities (global context).

Now, in view of our basic requirement on the size of each pass it is clear that tables of descriptions of the identifiers of the program can only be accomodated in translation passes which do not also require elaborate programs. There are essentially two such tables to consider: a table which holds the original form of the identifier (the actual letters and digits) to be used in matching the identifiers, and a table of descriptions of the kinds, types, and storage assignment, of the identifiers, to be used in generating the final code realizing the active parts of the program.

Identifier matching can in principle be made as soon as the microscopic structure has been established, but need not be done until the identifier description table is formed. However, in order to reduce the length of the intermediate language string representing the program it is advantageous in practise to do it early, i.e. before the analysis of the delimiter structure. Identifier matching is therefore done in pass 2. The fact that by this method the same identifier will be represented alike whether it denotes the same quantity, or, through the block struc-

ture, different quantities, does not cause any difficulty since in any case the association of identifiers with blocks is best postponed until the table of identifier descriptions is formed.

The table of identifier descriptions can only be formed after the analysis of the delimiter structure but must precede the processing and check of expressions. Because of the fact that ALGOL 60 effectively allows identifiers to be declared anywhere in the block in which they are local (labels) the establishment of the table of identifier descriptions of a block requires a scan of the complete block (Pass 4). Again, since the pass which performs the processing of expressions cannot be expected to hold the complete table, we must insert an additional pass, which serves only to distribute the information of the identifier description table to each place in the program where an identifier occurs (Pass 5).

In order to take best advantage of the block structure of ALGOL 60 programs, these two last mentioned passes are best arranged as a reverse pass followed by a forward pass. In this way the table of identifier descriptions need not be able to hold all quantities of the program simultaneously, but only the largest number of quantities which is available at any point of the program. We do not take full advantage of this possibility in GIER ALGOL but actually proceed as follows: During the reverse pass 4 the descriptions of identifiers are collected in a stack. Each time a block **begin** is encountered the top section of the stack is emptied into the program. During the forward pass 5 the identifier description table consists of a normal table having one position for each distinct identifier of the program plus a stack used to hold the descriptions of identifiers which have been declared to have different meaning. Each time a block **begin** is encountered the declarations of local quantities within the block are copied from the input into the table of descriptions. If the table already holds a description at a position which is about to be filled, the previous description is placed in the stack. Throughout the program the identifiers are now replaced by their full descriptions, such as they may be found in the table. When a block **end** is encountered the descriptions of quantities having the current block number are deleted from the table and previous descriptions restored from the stack. It will be noted that the use of a table having one position for each distinct identifier during the second of these passes actually will waste space. On the other hand we gain a great advantage in speed since the description of an identifier, given its number, may be found directly in the table. If a stack had been used for all currently active descriptions the retrieval of a description would need a search through the stack.

The above discussion covers the analyzing and checking stages of the compiler. The remaining part of the compiler generates the final machine code and places it in the correct locations on the drum tracks. In order to reduce the bulk of the logic which generates the one-address-instruction form of the program, pass 6, which performs the checking of types and kinds, also transforms the expressions into reverse Polish strings (Ref. 14). Pass 7 transforms this to a form which specifies the final machine instructions for arithmetic instructions while the final storage and internal references are still open. Pass 8 arranges the final machine instructions on the drum tracks and calculates the internal references. Finally, pass 9 rearranges the segments on the drum.

The tasks of the passes may be summarized as follows:

Pass 1. Analysis and check of hardware representation. Conversion to reference language. Strings are assembled.

Pass 2. Identifier matching. In the output each distinct identifier is associated with an integer between 512 and 1022.

Pass 3. Analysis and check of delimiter structure. Delimiters of multiple meaning are replaced by distinctive characters. Extra delimiters are inserted to facilitate the later scanning.

Pass 4. Collection of declarations and specifications at the **begin** of blocks and in procedure headings. Rearrangement of procedure calls.

Pass 5. Distribution of identifier descriptions. Storage allocation of variables.

Pass 6. Check of types and kinds of identifiers and other operands. Conversion to Reverse Polish notation.

Pass 7. Generation of machine instructions for expressions. Allocation of working variables.

Pass 8. Final addressing of program. Segmentation into drum tracks. Production of final machine code.

Pass 9. Rearrangement of the program tracks on the drum.

4.7. *Examples of translator action.*

The practical implications of the above division of the translation process into 9 passes may be explained by describing the fate of some of the language features through the process. These descriptions will give some examples of the contents of the actual byte strings produced by the passes. The individual bytes will be identified by bold-faced words, like delimiters in ALGOL 60. It may be noted that the complete al-

phabets of the output languages of all the passes are given in the appendix 2 of ref. 2.

The treatment of literal constants (strings, numbers, logical values) is dictated primarily by the run-time storage allocation of program. As already mentioned any constant used from one particular program segment track must be stored on that same track. Therefore constants must be carried through the translation process up to pass 8 at the place in the program where they occur. In other words, no table of constants for the complete program is ever formed. This treatment offers the advantage that during pass 7 constant operands may receive special treatment. In particular it is possible to evaluate any arithmetic expressions or subexpressions which deal entirely with constant operands. The individual classes of constants are processed as follows: Strings are assembled in pass 1. Those which are short enough to be stored in a single word are output as five bytes of which the four give the value of string when concatenated. Longer strings are placed in a list held on the topmost drum tracks and the drum track number and relative address replace the actual string in the five bytes communicated as output. Numbers are assembled in pass 3 and again are represented by five bytes. Similarly the logical values are output from pass 3 as five bytes. Constants pass unchanged through passes 4 and 5. In pass 6 the three types of constants are checked for compatibility with context. In pass 7 arithmetic expressions with constant operands are evaluated, but otherwise the constants pass through in the same form. Finally in pass 8 a constant table is built up separately for each program track. This table will also hold the program point descriptions necessary for local jumps from one segment to another.

Identifiers are assembled in pass 2. In the output from pass 2 each distinct identifier will be represented by an integer in the range 512 to 1022, irrespective of the block structure of the program. These integers pass unchanged through pass 3. In pass 4 the declarations of identifiers are collected whereas identifiers appearing in any other way pass through unchanged. In the output from pass 5 they are replaced by descriptions of four bytes each, giving the kind and type of the quantity, the block number, the block relative address, and where relevant an additional number specifying the number of subscripts, of parameters, or of switch elements, in the declaration of the quantity. In pass 6 the information on kind and type and on the number of parameters is used for checking and at the same time removed. The output from pass 6 describes the quantities only by three bytes, giving the block number and block relative address. In the output from pass 7 these operands can only

appear in association with bytes describing the operation parts of machine instructions. In pass 8 the operand descriptions are finally converted to machine addresses with the appropriate modifications (indirect, relative, indexed, etc.).

Simple expressions are structurally unchanged through the passes 1 to 5. In pass 6 the parenthesized expressions are converted to the Reverse Polish form in which the operands are quoted in exactly the same order as in the source text while the operators appear in the order in which they must act (Ref. 14). It may be noted that although this use of a Polish notation as an intermediate language is inspired by the techniques used by the Michigan school (Ref. 13) we do not see any advantages in the use of the quadruples suggested by Kantorovich (Ref. 15) because this notation introduces explicit names for working quantities which in many cases will have to be eliminated later. Our pass 6 operates in the manner described by Dijkstra (Ref. 7) and produces the Reverse Polish form, as defined by Hamblin (Ref. 14). In pass 7 the Reverse Polish form is converted into a form which makes explicit reference to the machine instructions and working locations. During this pass the order of references to operands is changed so as to minimize the use of working locations in so far as this is compatible with the possible side effects induced by references to procedures or formal parameters called by name. Thus, if a, b, and c are simple variables the expression

$$a \times (b + c)$$

will require the following instructions:

Take b
Add c
Multiply by a .

On the other hand, if f is a formal parameter called by name belonging to such a block level that it may affect the simple variable a the expression

$$a \times (f + c)$$

will give rise to the following instructions:

Take a
Place in working location
Take value of f
Add c
Multiply by contents of working location.

Conditional expressions are modified slightly during pass 3 so as to make the following processing more independent of context. Thus for example the following expression:

if *b* **then** *i* **else** *j* − *k*

will appear in the output from pass 3 as:

if-expression *b* **then-expression** *i* **else-expression** *j* − *k* **end-else-expression**

Here the bold-faced words stand for specific bytes in the output from pass 3. Thus while the input consisted of 8 bytes the output has 9 bytes. Also the ambiguous input symbols have been replaced by unique ones. A further differentiation of conditional expressions is produced by pass 6 where types are recognized and checked. In the output from pass 6 the input symbol **else-expression** is replaced by either **else-*RF*-expression**, **else-*R*-expression** or **else-address-expression** according to the type of the expression (arithmetic, Boolean, or designational). A similar differentiation is made for **end-else-expression.** In pass 7 these distinctions are taken properly into account in generating the instructions terminating the subexpressions and consequently the output from pass 7 does not make the distinctions, but only produces bytes representing **else** and **end-else.** Finally in pass 8 these bytes are used to form the proper jump instructions as follows. Pass 8 in its reverse scan first encounters **end-else.** This causes two copies of the current program point to be recorded in a stack of program points. When subsequently **else** is encountered, one of these is changed to point to the current point, to be used by **then**, and then an unconditional jump instruction to the point recorded in the other item of the stack is output. Similar actions are performed at **then** and **else-if** (an **if** immediately following an **else** in the source program). By this technique the use of conventional symbolic addresses becomes unnecessary.

Assignment statements are first processed in passes 3 and 4 which make the structure more explicit. As an example the following assignment statement:

$$a := b := c := d - e$$

will appear in the output from pass 4 as

a **first-** **:=** *b* **:=** *c* **:=** **prepare-assign** *d* − *e* **end-assign**

where the four bold-faced combinations represent unique output bytes from pass 4. By this transformation the type checking during pass 6

becomes greatly simplified. In the output from pass 6 the assignment is treated like an operator in the inverse Polish string. If in the above example a, b, and c, are of type **integer** while d and e are of type **real** the output will be

$$abcde - \textbf{round prepare-assign} :\!= :\!= :\!=$$

where of course each of the operands is represented by the proper description in terms of block number and block relative address.

Procedure statements and function designators are made explicit by pass 3 like assignment statements. In pass 4 they are transformed in such a manner that they appear as a call which includes a listing of each actual parameter, followed by the code representing any actual parameters which are compound expressions. This is quite close to their final form except in the case of calls of standard functions having one parameter called by value (sin, cos, etc.). These are converted to appear like operators in the final program by pass 6.—An example of the transformation of procedure calls is given below.

In generating internal program points the translator nowhere employs the customary symbolic address technique. Instead the known structure of the text, in combination with stacks, is used to transmit the static descriptions of program points from the place in the program where the points appear to the place where the description is needed. This may be illustrated by the following example of a procedure call:

$$P(a+b, c, d-e, f)$$

In the output from pass 3 this appears as 14 bytes:

P **begin-call** $a+b$ **call-parameter** c **call-parameter** $d-e$
call-parameter f **end-call**

Pass 4 transforms this to the following string of 18 bytes (generated in the reverse order):

begin-call P f **expression** c **expression end-call begin-expression**
$a+b$ **end-expression begin-expression** $d-e$ **end-expression**
bypasslabel

At this stage the list of actual parameters has been formed. This list reverses the order of the actual parameters. This is convenient for pass 8 which has to replace each occurrence of the symbol **expression** by the static description of the entry point of the corresponding expression. In fact, the reverse pass 8 need only record the program point description of each **begin-expression** in a stack and then use the top element of this

stack to replace each occurrence of the symbol **expression**. In a similar way the symbol **bypasslabel** serves to record the return point of the call during pass 8. Since this will be used to replace the symbol **end-call** it is clear from the above structure that a second stack must be used during pass 8.

5. Translator methodology.

As will be clear from the discussion of the background of the project the development of new methods of translation had a low priority. Even so we have in the following notes tried to describe the most important tools employed in the project. Where this description differs from similar descriptions given elsewhere it will probably be found that the difference is more one of the way we think about a tool than in the way we use it. However, in view of the present rudimentary state of the discussion of translation methods this may still be of interest.

5.1. *The stack as a transmission device.*

The well-known principle of a stack (or push-down list, or cellar, or first-in-last-out-list) is used extensively in the GIER ALGOL compiler. We like to regard the stack as a device for transmitting information from one point in a text to another. It may be used wherever the sections of the text within which a piece of information to be transmitted exists, form a nested parenthesis structure. This means that within one communication process it is necessary to have as many stacks as there are independent parenthesis structures to be taken care of.

A further distinction may usefully be made between the case where a piece of information transmitted through the stack will be used just once, and the case where one such piece may be used any number of times. We tend to denote the former type of stack as a true push-down list, the designation "stack" being used for the latter type.

The following notes discuss briefly the stacks used throughout the compiler.

Pass 3. A push-down list holds those delimiters having left-parenthesis character which have not yet been matched by a corresponding right-hand delimiter. Example: While scanning the following text:

begin $a := $ **if** b **then** $c[i]$ **else** d;

the stack is used to communicate as follows:

Transmitting symbol	Receiving symbol
begin	;
:=	;
if	**then**
then	**else**
[	]
else	;

The information transmitted through the stack is used to create explicit right-hand delimiters corresponding to those left-hand symbols which do not already have them, and to replace those delimiters which in ALGOL 60 are used for more than one purpose by more explicit ones. This is reflected in the following text which is the output from pass 3 of the above input text:

begin $a :=$ **if-expression** b **then-expression** $c[i]$ **else-expression** d **end-else-expression end-assign**

Pass 4. This reverse pass uses a push-down list to transmit declarations of quantities from the place in the program where the declaration is written to the **begin** of the corresponding block, and also to transmit the information about each actual parameter to the left parenthesis of the procedure call.

Pass 5. A push-down list is used to transmit the descriptions of identifiers across any such sections of the program where the identifier is declared to have another meaning. The push-down list is used side by side with a normal list having one position for each distinct identifier of the program holding the current description of that identifier.

Pass 6. This uses two push-down lists. An operator list transmits the operators from the place where they occur in the parenthesized input string to the place where they are needed to form the corresponding Polish string. An operand push-down list transmits the descriptions of the kinds and types of operands (identifiers, numbers, variables, etc.) to the place where they are operated upon.

Pass 7. A stack transmits the description of the storage of each operand from the place in the Polish input string where it appears to the place where it is finally operated upon, usually forming the address of an instruction.

Pass 8. Two push-down lists are used, both for transmitting references to points in the program from the place where the point is to the place where it is referred to.

Running program. A stack is used to communicate all variable parts

of the program. These include declared and internal working variables, storage allocation coefficients of arrays, dynamic program point descriptions, and block information.

5.2. *The Turing machine approach to text analysis.*

In GIER ALGOL syntactic analysis of texts is mostly carried out by a process which has some similarity to the working of a Turing machine. The crucial feature of this method is the insistence on an explicit enumeration of the states in which the scanner may find itself at any time. These states are regarded as equivalent and each is characterized by its number. It is understood that the current state number must give a description of the situation which is sufficient to be used for a complete determination of the action to be taken on the following symbol in the input, including the assignment of a new current state. Indeed, this requirement may be regarded as the criterion that a proposed set of states is in fact adequate for the analysis at hand.

This approach may conveniently be illustrated by the logic necessary to analyze and check a text which purports to contain a number in the sense of ALGOL 60. Let the text be composed of digits, points, exponent tens, plus and minus signs, and a further class of symbols which we shall denote terminators. Further let us set out to scan a piece of text which contains any number of terminators followed by one ALGOL 60 number followed by one terminator. In order to analyze and check such a text our scanner will need 8 states. The logic may conveniently be described by the following table which for each combination of a state and an input symbol gives the new state. When an error has been found the new state *A* (alarm) is indicated.

State	+ −	digit	point	ten	terminator
1: Only terminators yet	2	3	4	6	1
2: After first sign	*A*	3	4	6	*A*
3: Among digits before point	*A*	3	4	6	integer
4: Following point	*A*	5	*A*	*A*	*A*
5: Among digits following point	*A*	5	*A*	6	decimal number
6: Following ten	7	8	*A*	*A*	*A*
7: After exponent sign	*A*	8	*A*	*A*	*A*
8: Among exponent digits	*A*	8	*A*	*A*	number with exponent

The sequence of states of the scanner in a few examples is given below:

Input symbols: ; − 2 0 ;
States: 1 1 2 3 3 integer

Input symbols: $_{10}$ + 4 3 ;
States: 1 6 7 8 8 number with exponent

The above description has stressed the checking aspect of the Turing machine approach. However, an equally important aspect is the ease with which arbitrary actions may be specified. In general each point in the table with arguments state and input symbol will correspond to exactly one unconditional action. By using this approach it is usually possible to avoid tests in the individual actions to a surprisingly high degree. This in our experience is a very effective way of reducing the bulk and execution time of the translator algorithms.

The most extensive application of this approach is made in pass 3. Surprisingly we have found that only 32 states exist while scanning a correct ALGOL text in the manner suggested by Dijkstra (Ref. 7), i.e. scanning up to the next following delimiter in each process. In order to take care of source program errors we need another 4 states.

5.3. *The use of tables and switches.*

A large part of the logic of the GIER ALGOL translator is described in tables. Usually the words in the tables contain 4 parameters of 10 bits each. Typically the basic translation cycle of a pass is controlled by a central administration which takes an input byte, uses it to look up a table, unpacks the word found in the table, and jumps to the action whose address is given in one of the parameters of the word. The remaining 3 parameters may then be used freely by the action program as output byte values or the like.

This method is partly dictated by space economy considerations. However, it has several positive advantages. First, it contributes to the clarity of the logic. But the most important feature is the insistence on the use of a switch for controlling the logic. We see this use of a switch as an alternative to expressing the translator logic as an algorithm which includes numerous tests of relations and Booleans for controlling the action. We feel that the switch method is superior in all respects: clarity, speed, storage economy.

5.4. *Pseudo-evaluation of expressions.*

The basic method used in the processing of expressions is pseudo-evaluation. This method is in fact used twice: for type checking (pass 6) and for the generation of final machine instructions (pass 7). By pseudo-evaluation we mean that the compiler scans the expressions while keeping at all times a complete account of all currently active operands and using this account to combine the operands in the same manner as they will be combined during the actual evaluation at run time. The pseudo-

evaluation differs from an actual evaluation in the kind of information about the operands which is being processed. During actual evaluation we operate on the values of the operands. The pseudo-evaluation performed during pass 6 operates on descriptions of the kinds and types of the operands. During pass 7 we operate on the descriptions of the storage of the operands.

The pseudo-evaluation is carried out at a stage when the expressions are written in the Reverse Polish form. In this form there are two kinds of operands: explicitly named variables and anonymous variables in the stack. The appearance of a named variable always means that this variable should be transferred to the top of the stack. The operators always operate on anonymous variables.

The use of this method results in a very transparent logic. At the same time it allows a great flexibility in the kind of operators which may be handled. This is particularly important in the generation of machine code for the more intricate structures of ALGOL 60 such as the array declaration. Likewise the individual treatment of a great variety of operands is possible. This is important if a complete checking of types is aimed at.

5.5. *Operand descriptions.*

In using the method of pseudo-evaluation the form of the operand descriptions is of some importance. Typically, during type and kind checking the action program of each operator will have to check that the operand, or operands, described by the top items of the operand stack conform to proper usage. In ALGOL 60 we have to distinguish between 25 classes of operands while the number of operators is rather larger. A simple way to handle this large number of combinations is to describe each class of operands as a Boolean vector, kept as a bit pattern in one machine word. Each element of this Boolean vector is designed to characterize that division of the total class of operands which is of interest in one particular operator action program. In this way the action program of a particular operator will only have to test one single bit of the operand description found at the top of the operand stack in order to check whether the operand is proper.

In using this method we found that we needed 35 bits in the operand description, which could therefore easily be kept in one machine word.

5.6. *Formal errors in the source program.*

One of the principal aims of the design was to let the compiler be an effective checker of source programs by being able to continue to process

a program even after errors in it had been found. The clue to this problem was found to lie in a complete integration of the logic of translation and that of treating errors.

This approach could be applied most simply to errors of compatibility of the declarations of an identifier and its uses (missing declarations, incompatible kinds or types). When such an error is found the description of the identifier is changed to denote a special internal quantity, "undeclared", and a suitable alarm message given. In all later occurrences the identifiers of this description are accepted without further alarm messages. This method suppresses the avalanche of error messages which will often occur in compilers with a more primitive technique of error handling.

Errors of the delimiter structure cannot be handled as effectively as this. The problem is to reestablish the synchronism between the text given in the input string and the variables describing the state of the translator. In general this problem has no unique solution and it is necessary to resort to some ad-hoc convention. The solution we have adopted consists in the complete removal from the input text of the surroundings of the sore spot. This surrounding will usually comprise a basic statement. In particular, care has been taken never to remove any **begin** or **end** from the input string since this would usually cause a very great amount of additional difficulty. This removal process can be accomplished very easily during the forward pass 3, which detects the errors of the delimiter structure, and the reverse pass 4.

Every error message produced by the translator gives the number of the line in which the error occurs, in addition to a characterization of the error. In this way only the carriage return characters of the source program need be kept through the translation passes, while the original identifiers can be eliminated completely during pass 2. In order to facilitate the identification of errors pass 1 will optionally produce an output of a copy of every 10th line of the source program with the line number attached.

5.7. *Writing and checking of the translator.*

During the planning and development of the logic of the translator ALGOL itself was used extensively. However, no attempt was made to use a boot-strapping method, for two reasons: first, the size of the available store would prohibit such an approach. Second, in our experience the manual transcription of the algorithms into machine lan-

guage is an effort which pays off very well in the improved speed and storage economy of the final product. The second of these statements may be illustrated by the fact that pass 6, which had required one to two man months for the development of its logic, could be transcribed into machine code in about one man week.

During the development of the translation passes some pains were taken to make all the members of the group familiar with all the logic. Indeed, we believe that only by distributing the knowledge of the over-all design of the translator as widely as possible among the participants in the work it is possible to derive the maximum of benefit from the freedom to choose the most suitable pass to perform each translation process. One of the ways to achieve this was to insist on a very thorough manual testing of all parts of the logic, before each part of the translator was loaded into the machine for the first time. This manual checking would always be done by another person than the one who had done the actual programming, and would therefore force at least one other person to familiarize himself with all details.

At all stages of the design great stress was placed on the problem of checking the translator. The goal is to obtain the maximum of convenience in obtaining information on the performance of the translator passes and in specifying the test input. The solution adopted was the following: The test output consisted entirely of the values of the bytes produced by each pass. These values were printed neatly as ten integers per line by a short output program which is a part of the general pass administration. Only in the case of pass 8, which produces machine code, another test output program had to be used. The problem of test input, on the other hand, was solved simply by checking the passes strictly in their natural sequence, in other words by checking first pass 1, then pass 2, etc. Therefore the test input to any pass is written in ALGOL and the check-out problem becomes one of writing test programs in ALGOL specially taylored to each pass to make sure that all parts of the logic of the pass are properly put to work.

This method proved to be highly successful. About 160 short ALGOL programs were needed to check the complete translator while no other diagnostic facilities (memory dumps, traces, or the like) were found necessary, or even desirable. Even so the passes 6, 7, 8, 9, and the running system, including standard procedures, were all loaded into the machine for the first time and checked during a period of 20 days. Although there were still errors left after this period the result was good enough to be distributed and widely used for several months in most GIER installations.

6. Size and performance.

6.1. *The size of the translator and running system.*

During translation the system occupies 138 out of the 320 tracks on the drum, the remaining tracks being available for the partially translated program. The number of words used for the programs and permanent tables and the use of the remaining part of the core store during each phase of the translation and execution process are as follows:

Phase	Words for Program	Words for Tables	Use of the remaining part of the core store
General pass administration	371	0	Each pass has 769 words at its disposal.
Pass 1	501	132	Buffer for long text strings: 40 words.
- 2	89	62	Identifier table: max. 618 words.
- 3	264	268	Delimiter push-down list: max. 237 words.
- 4	216	43	Declaration collection push-down list: max. 510 words.
- 5	172	29	Identifier description lists: max. 570 words.
- 6	348	299	Operator push-down: max. 50 words, operand push-down: max 72 words.
- 7	529	179	Operand stack: max. 61 words.
- 8	336	140	Program reference push-down lists: max. 293 words.
- 9	185	16	Track table: 182 words.
Running system	280	0	Program segments and variables: max. 810 words.
Standard procedures	740	0	
The total system	4031	+ 1168 = 5199 words.	

The size of the intermediate forms of average programs varies by a factor of about 2, the output from pass 2 being the shortest and the final machine code the longest. This means that if the final program can be accomodated in the machine then the same will hold for all the intermediate versions.

The capacity of the working areas of the passes is adequate. In fact, although the compiler is used extensively for large production programs in several installations we know of no realistic program which could not be compiled because of overflow of push-down lists or stacks.

During program execution only the running system and standard procedures are needed and the pass programs and tables may be overwritten by variables. However, if the translator is not destroyed in this

way the translation of a program may start immediately upon completion of execution of the preceding program.

6.2. *The performance of the system.*

As already mentioned the compile time is about 2 seconds per final segment of 40 words, corresponding to about 60 one-address machine instructions. For very short programs a basic time of about 4 seconds becomes prominent. This is about 4 times as slow as input from binary tape, but faster than input of symbolic machine code. At this speed the compile-and-go mode is very attractive, and in fact no program for producing an output of the compiled program in binary form has so far been written.

More than one third of the translate time is spent in pass 1. This pass is in fact input limited in speed. Each of the remaining passes takes between 5 and 14 percent of the total translate time of fair-sized programs, pass 2 (identifier matching) leading with pass 7 (generation of machine instructions for expressions) a close second.

The error detecting capabilities of the translator fully satisfy the design goal and have proved very convenient in practise. In fact, the translator may be used as an extremely effective proof-reading device.

When the performance of the translated program is being discussed the normal reaction is to ask for a comparison of the execution speed of a compiled and a hand-coded program. There are several reasons why we do not wish to try to answer this type of question:

1. The question is undefined as long as it is not specified what the kind of the problem is, who does the machine coding, and what the rules for transcribing the algorithmic constituents between ALGOL 60 and the machine coding are supposed to be. In particular on the last score there is room for a large factor of uncertainty in a system like GIER ALGOL which includes powerful mechanisms for storage allocation not normally considered in hand coding.

2. Even if a well-defined comparison can be made the outcome of it is of no particular interest because it does not point to constructive improvements of the design.

For these reasons we wish to base our assessment of the performance of the system on analyses of the time spent on the various language functions during the execution of realistic, practical algorithms. By this approach we will be able to detect the bottlenecks of the execution. Analyses of this kind have been made for algorithms for inverting ma-

trices, for finding eigenvalues of symmetric matrices, and for definite integration by Simpson's rule. These analyses very definitely point to two bottlenecks: (1) subscripting, and (2) transfer of control to a segment which is already present in the core store. The relative importance of these two items varies greatly, not only with the program, but also with the manner in which the program happens to be segmented. However, as a rough average it appears that these two items together account for well over half the execution time of many realistic programs. As a further conclusion it may be stated that such programs might be speeded up by a factor of two or more if two or three special machine instructions designed to take care of these two problems were included in the machine.

Acknowledgements.

The design of GIER ALGOL which goes beyond the sources already quoted in the historical notes is due to a day-by-day pleasurable and inspiring collaboration of Jørn Jensen and the present writer. Where the large-scale design is concerned it is impossible to disentangle our contributions. The machine coding was done almost entirely by Jørn Jensen. While the project was in progress we were joined by Peter Kraft, Henning Christensen, Paul Lindgreen, Knut-Sivert Skog, and Peter Villemoes, who did a large amount of the practical and clerical work while at the same time learning the techniques. The project was also supported in various ways by several other members of the staff of Regnecentralen, in particular Agnes Michaelsen and Kirsten Andersen.

Finally the debt to Niels Ivar Bech, director of Regnecentralen, should be recorded. In fact, his continued and enthusiastic support was an indispensable condition for the success of the project.

REFERENCES

14. Hamblin, C. L., *Translation to and from Polish notation*, Computer Journal 5 (1962), 210–213.

15. Kantorovich, L. V., *On a mathematical symbolism convenient for performing machine calculations*, Doklady AN USSR 113 no. 4 (1957), 738–741.