BIT 3 (1963), 124–140

THE DESIGN OF THE GIER ALGOL COMPILER
PART I

PETER NAUR

Abstract.

The report gives a full description of the design of an ALGOL 60 system for the GIER, a machine having 1024 words of cores and 12800 words on drum. An introductory section gives the historical perspective of the design and the principal characteristics of the machine. The second section discusses the problems of the running ALGOL program: storage allocation and addressing of variables, program references, procedure calls, and the automatic administration of transfers of program segments from the drum to the core store.

The second part of the report will describe the translator, and the performance of the system.

Contents.

1. Introduction.

GIER ALGOL is an ALGOL compiler written for the medium size computer GIER manufactured by Regnecentralen, Copenhagen.

The compiler was first distributed in a slightly restricted form in September 1962. A final version was distributed in February 1963. By

this time it had become the dominating programming language for the machine in most of the installations.

The language of GIER ALGOL is ALGOL 60 except for the omission of integers as labels, arrays called by value, and own arrays. Input and output (including transfers of variables between the magnetic drum and the core store) are controlled by means of standard procedures. A full description of the language is given in the published manual (Ref. 2).

In the following the design of the system and compiler is discussed in some detail. Some information on the performance is also given. Since the purpose is to give a clear picture of the over-all design the discussion is not confined to the points where the system deviates from designs described elsewhere in the literature.

2. Background of the project.

2.1. *Historical notes.*

The GIER ALGOL design has been influenced by a variety of other projects. To put the description in its proper perspective it is helpful to give a brief historical account of the development of our ideas during the years 1959 to 1961.

Our first exposure to compiler techniques came from Professors F. L. Bauer and K. Samelson of the University of Mainz, Germany, the leaders of the ALCOR group. In June 1959 P. Mondrup and W. Heise of Regnecentralen spent a few days at Mainz and had the opportunity of studying the sequential techniques developed there (Ref. 9). These techniques were adapted to our machine DASK by P. Mondrup and form the main frame of the DASK ALGOL compiler completed in late 1961.

In the meantime we became increasingly aware of the problems inherent in the ALGOL procedure concept and of the incompleteness of the description of procedures given in the Zürich ALGOL report (ALGOL 58). We raised this problem within the ALCOR group in December 1959, but were disappointed to find that the other members of this group did not seem prepared to take a common stand in the problem. Fortunately the problem was cleared in ALGOL 60 and during the time which followed we developed the scheme for handling the ALGOL 60 name concept which was used in DASK ALGOL (Ref. 3).

In March 1960 we opened an active personal contact with Prof. A. van Wijngaarden, Dr. E. W. Dijkstra, and Mr. J. A. Zonneveld of the Stichting Mathematical Center of Amsterdam, Netherlands, who came to spend a few days of informal discussions with us. These discussions showed that both groups independently had arrived at the same conclu-

sion, viz. that the design of the system which administers the run-time requirements of the ALGOL program (storage allocation and procedure calls) must be considered the primary problem while the translator is a secondary problem.

The Dutch group impressed us greatly by their very general approach. However, although they were prepared to put their solution of the problem of recursive procedures at our disposal we decided to stick to the more modest approach which we had already developed to some extent. The reasons for this reluctance were practical. First of all we felt the size of the problem to be already rather frightening, in particular because we still had to face the problem of the two levels of store in DASK, and also because we wished to include far more error detecting capability in our compiler than the Dutch. Also at that time we feared the loss of running speed of a system which included recursive procedures (a fear which we now know was unfounded).

The first news of the success of the Dutch project, in June 1960, fell like a bomb in our group. However, we quickly regained spirits and during July to September completed the design and coding of the DASK ALGOL running system, including run-time facilities for drum transfers of program and variables (Ref. 4) and the set of standard functions and input and output procedures (Ref. 5). This system was wired into DASK as a new fixed store and was completed in this form in January 1961.

In the meantime the coding of the DASK ALGOL compiler itself had proceeded, although at a much slower speed than anticipated. As already mentioned it is based on the Mainz principles, but the adaptation to DASK had required important modifications. Essentially it is a three pass compiler, using three tape units, one of them holding the compiler, the two others being used to store the partially translated program. The first pass performs a rather modest preprocessing, mainly taking care of the peculiarities of the paper tape hardware representation. The second pass performs almost all the translation, keeping all its tables on the drum. The third pass completes the addressing, which is a very quick process. This compiler was not completed until about September 1961.

Already during the later phases of the coding of DASK ALGOL other approaches were considered, as a preliminary to coming attempts. When the idea of the syntax-directed-compiler by Irons (Ref. 6) was published some of us got very excited about it. From about December 1960 to June 1961 we had Mr. B. Mayoh working on using this for a compiler for the GIER which was then in its last phases of development. However, after this trial we decided that, as far as we were concerned, the approach was a mistake.

Other contacts included a visit of the present writer to Amsterdam in April 1961 during which Edsger Dijkstra supplied me with all desired details of their method of addressing at run time and of scanning the source program (Ref. 7). Another decisive influence came from the Storage Allocation Symposium in Princeton in June 1961 where Jørn Jensen learned about the dynamic storage allocation scheme planned for the Ferranti Atlas (Ref. 8). Finally, during a stay at the University of North Carolina, Chapel Hill, during the later half of 1961 the present writer had the chance to reconsider the complete problem of translating ALGOL 60. The most important result of this was an approach to the problem of analyzing and checking the source text, which although related to well-known methods, in particular Grau's version of the Bauer–Samelson method (ref. 12) yet has some merit of its own (see the Turing machine approach below). A further result of this visit was the exposure to the idea widely used in the U.S. of using a Polish notation as an intermediate form of expressions during translation (ref. 13).

This takes us to January 1962 when it was decided that a GIER ALGOL compiler should be written.

2.2. *Characteristics of GIER.*

The GIER is a machine manufactured by Regnecentralen. It has also been marketed as DISADEC. The following figures refer to the minimum configuration for which the ALGOL compiler was primarily designed. For further details see ref. 1.

Stores. Core store: 1024 words of 42 bits, access time 8.8 microseconds. Magnetic drum: 320 tracks of 40 words each. Transfer time of complete track: 20 milliseconds. During drum transfers other operations may take place in the machine.

Word structure: Of the 42 bits two are regarded as marks and are not processed in parallel with the rest of the word. Floating point operations divide the word into a 10 bit exponent and a 30 bit mantissa while the marks are irrelevant.

Order structure: The marks in each word select the order interpretation modes: (a) One instruction of 40 bits or two instructions of 20 bits in the word. (b) Fixed or floating operation.

Addressing facilities: There is one normal index register, the p-register, and an index register which is also coupled to the subroutine return mechanism, the s-register. In addition the address may be relative to the order counter, r-relative addressing. Indirect addressing is provided. Instructions of 40 bits may also include an incrementing of the address.

Operation times: Fixed point operations range from 36 to 50 microseconds. Floating-point operations: addition 100 microseconds, multiplication 170 microseconds, division 220 microseconds.

Input: 8-hole paper tape, reading at 500 characters per second.

Output: 8-hole paper tape, punched at 150 characters per second.

2.3. *Aims and methods.*

The aims of the project were roughly as follows:

1. GIER ALGOL should be a practical working system, taking full advantage of the machine as far as this is compatible with (a) the generality of ALGOL 60 and (b) a dead line on the completion of a workable compiler of September 1, 1962.

2. It should include virtually complete error detection of the source program, and should be capable of finding any number of errors in one compilation run.

The methods adopted for achieving these aims are as follows:

1. The design centers around a dynamic storage allocation of variables in a stack, basically similar to the design of Dijkstra (Ref. 7 and 10).

2. The storage of program is handled by an automatic administration of transfers of program tracks to the core store at run time, completely integrated with the stack administration.

3. The design of the translator again is based on storage allocation considerations. These indicate that to achieve speed many internal passes should be used, employing the drum for storing the partially translated program, while the program and tables of each pass should be small enough to be stored completely in the core store.

4. During translation, error handling is integrated with the translation, i.e. the occurrence of an error is not considered the exception but will be handled by the same kind of logic as is used for any regular language feature.

All these methods are discussed in greater detail in the remaining part of the present report.

3. The running system.

The running system is the fixed administration which is used by any translated ALGOL prgram while the program is executed. It occupies the last 200 words of the core store and its most important tasks are the handling of the dynamic storage allocation and the execution of procedure calls.

The running system must be defined before the translator is written

since it is an integral part of the definition of the target language. Our previous experience shows that the proper order to decide on the solution is: storage allocation, addressing, procedure call.

3.1. *The non-homogeneous store.*

In a machine with a core store of 1024 words of 42 bits and a backing drum of 12800 words some method of transferring the translated ALGOL program and its variables between the two media must be found. In DASK ALGOL a system based on explicit information supplied by the programmer was used for both program and data (Ref. 4). In GIER ALGOL the transfers of variables are again fully under the control of the programmer, although the tools placed at his disposal, standard procedures, are very different from those incorporated in DASK ALGOL. Transfers of program in GIER ALGOL, on the other hand, are done automatically by the system.

The decision to provide automatic transfers of program, but not of variables, is based on the following considerations:

1. The programmer will be fully aware of the storage demands made by the variables of his program, but will have only a very inaccurate knowledge of the length of the code needed to represent the algorithms. Therefore the programmers task of specifying the transfers of variables is much easier than the specification of transfers of program.

2. Transfers of variables to and from drum may be regarded as input/output operations and will therefore be analogous to operations which the programmer will have to learn and use anyway. Any conventions for transfers of program sections will constitute a unique addition to the conventions which the programmer will have to learn.

3. In a sensible system the machine instructions will be constant throughout the run of the program. For the implementor the transfers of program are simpler because no question of saving a piece of program which has been copied from drum to cores ever arises.

3.2. *Storage and addressing of variables.*

Since all variables which can be referenced directly by the ALGOL program will be stored in the core store, the familiar stack arrangement can be used (Refs. 7, 10). This is inherently a very economical storage principle. However, to make full use of the economy of the arrangement, the scheme chosen for utilizing the storage space left unused by the stack must be able to follow the variations of the size of the stack during the

run of the program. This was a decisive factor in the design of the system for allocation of program described below.

In the addressing system developed by Dijkstra (Refs. 7, 10) two kinds of variables are distinguished: (a) Named variables, addressed through their block number and block relative address; (b) Anonymous variables, addressed relative to the current top of the stack. In Dijkstra's system and in DASK ALGOL (Ref. 4) anonymous variables are used wherever the identity of a variable is given completely through the structure of the ALGOL text. In a machine having built-in floating point operations, but no special facility for working at the top of a stack, like GIER, references to named variables can usually be performed much faster than references to anonymous variables (see below). For this reason the GIER ALGOL translator has eliminated references to anonymous variables as far as possible. This is achieved by replacing anonymous variables by internally named local quantities. This will cause a certain slight waste of working locations, since each block will have to reserve as many locations as are used at any one point of it. The only case where this replacement of anonymous variables by local ones is not possible is that in which an anonymous quantity is handed over from one block to another and therefore must be addressed relative to the universal stack top pointer. The simplest example of this is the return information which is generated whenever the program transfers control to a procedure or a formal parameter called by name which is equivalent to a procedure. The return information is used by that procedure to transfer the control back again (see the section on program points below).

The basis of the addressing of named variables is the recognition that at any moment during the execution of a program the variables which may be referenced directly are exactly those which are local to the youngest incarnations of the lexicographically enclosing blocks. The relative addresses of the variables within a block are finally calculated during translation. In order to obtain the absolute address of a variable we only have to add what we call the "stack reference" of the corresponding activation of the block to the relative address. The absolute addresses of all accessible variables may therefore be calculated if the program has available the block numbers and relative addresses of the variables on the one hand, and the "stack reference" of the youngest incarnations of the enclosing blocks on the other. These latter are held in the table called the DISPLAY (Ref. 10). In an obvious notation we have for a variable described by its block number and relative address:

$$\textit{stack reference} = \textit{DISPLAY}[\textit{block number}]$$
$$\textit{absolute address} = \textit{relative address} + \textit{stack reference}$$

In the final machine code it is more convenient to replace the block number by the so-called "DISPLAY reference" which is the absolute address of the location which holds the stack reference. Therefore we have

$$stack\ reference = store[DISPLAY\ reference]$$

This scheme requires that the DISPLAY be up-dated whenever the control changes to a different block environment. The basis for doing this is given below in the section on program points.

In a machine having built-in floating point operations, like GIER, the address calculations according the above scheme must be done by some very direct method if the use of an undesirably high proportion of the running time for this purpose shall be avoided. The solution used in GIER ALGOL consists in using the two index registers as follows:

a) Variables in the outermost block. For these variables the stack reference will be constant throughout the program and the compiler will be able to calculate the final absolute machine addresses.

b) Variables in the currently local block. The stack reference of the local block is at all times available in the index register p, as well as in the appropriate element of the DISPLAY. References to local variables can therefore be made by means of p-relative addresses.

c) Variables in intermediate blocks. These are addressed relative to the index register s. The s-register is used for other purposes as well. However the compiler is able to keep track of these uses and will be in a position to insert explicit machine instructions of the form

$$s := store[DISPLAY\ reference]$$

where necessary.

The decision to use the index registers in this way is based on the observation that in practical programs, in particular published procedures, the overwhelming majority of identifiers refer to local quantities.

3.3. *Program points.*

No part of the translated program can be executed unless the DISPLAY is up-dated so as to contain the stack references of the youngest incarnations of the lexicographically enclosing blocks. The up-dating is performed on the basis of the chains of stack references which form a part of the block information stored in the stack at all block-relative addresses 0 and 1. In fact, each time a block is entered the DISPLAY reference corresponding to its block number and the stack reference of the youngest incarnation of its innermost enclosing block are placed at

relative address 0 of the newly reserved section of the stack. By these means an up-dating of the DISPLAY to make it correspond to the newly established environment may be performed at any time while this incarnation of the block is still active if only the value of the stack reference is supplied. The algorithm is as follows (I owe this improved form of the arrangement, which only uses one input parameter, to a personal communication from E. W. Dijkstra in February 1962):

```
j := DISPLAY reference part(store[stack reference]);
for sr := stack reference,
          stack reference part(store[sr]) while
          j ≦ DISPLAY reference of block 0 do
    begin store[j] := sr; j := j+1 end;
```

An alternative description which lies closer to the realization in GIER ALGOL is the following:

```
    switch UPDATE := upd0, upd1, upd2, ...;
    sr := stack reference;
    go to UPDATE[1+DISPLAY reference of block 0-
                 DISPLAY reference part(store[sr])];
    ...
upd2: DISPLAY[2] := sr; sr := stack reference part(store[sr]);
upd1: DISPLAY[1] := sr; sr := stack reference part(store[sr]);
upd0: DISPLAY[0] := sr;
```

The switch UPDATE must have as many elements as there are lexicographically enclosing blocks in the program. This number is determined by the compiler.

When the control is transferred from one point in the translated ALGOL program to another it is in general necessary to up-date the DISPLAY to correspond to the new environment. Consequently, in order to specify a transfer of control we must in general supply both the static description of the destination (segment track number and relative address, see the description of program allocation below) and a dynamic description of its environment, the stack reference. This set of three numbers, track number, track relative address, and stack reference, together define what we call a program point.

Transfer of control within the same block need not use the general program point specification since the DISPLAY remains unchanged. This state of affairs may easily be recognized by the translator in the practically must important cases, jumps created by if and for clauses.

In GIER ALGOL extensive use is made of this possibility of simplifying the execution.

3.4. *The local declaration.*

Program points are treated as a kind of variables. Every explicitly named point of a block (labelled points and entries into the bodies of local procedures) will have a location in the stack allocated to it, like the location where the value of a variable is kept. When the block is entered all the locations of program points are initialized. This is done by a subroutine call, the so-called local declaration, placed at the **begin** of each block. The local declaration contains information on how may locations must be reserved in the stack and gives the static description of each named program point.

Program points which are essentially similar to the explicitly named one are generated in procedure calls, as described below.

Switch declarations are also taken care of in the local declaration, but shall not be described in detail here.

The complete machine formats of the local declaration are given in ref. 2, appendix 2.

3.5. *Descriptions of quantities.*

The use of locations in the stack for holding the descriptions of named program points is one case of the more general use of descriptions of quantities which extends to array and switch identifiers.

The description of an array identifier comprises (1) an absolute address which points to the point in the stack where the so-called "dope vector" is stored and (2) another absolute address which is a base address of the elements. All array identifiers of the same array segment share the same dope vector. Array identifier descriptions are placed in the stack by the explicit code representing the array declaration.

A switch identifier is described by a word giving (1) the absolute address of the first word of a list of the elements kept in the stack and (2) the number of switch elements. The switch identifier descriptions and the descriptions of the associated switch elements are placed in the stack by the local declaration.

An important factor in the choice of the formats of identifier descriptions is the requirements of the procedure call. These requirements also lead to the introduction of descriptions of simple variables, as explained more fully below.

3.6. *Procedure calls.*

The central problem of the procedure call is to communicate to the procedure body a sufficient amount of information about the actual parameters. This information is transmitted via locations in the stack, one for each parameter, the so-called formal locations. In detail GIER ALGOL proceeds as follows:

The translator has transformed each procedure call into a call of a subroutine followed by words describing the actual parameters and in addition giving the static description of the point following the complete call, the return point. A last word gives the DISPLAY reference and block relative address of the location in the stack which holds the description of the entry point of the procedure being called. The possible forms of words used as parameters (see also ref. 2, appendix 2) and their treatment in the call are the following:

1. Constant.

The value of the constant is given and is transmitted to the formal location.

2. Described in stack.

The call supplies the DISPLAY reference and relative address of a location in the stack. This location will hold a description of a quantity (see above). In the call the description is copied into the formal location. This form is used for array, switch, and procedure identifiers, and for formal parameters.

3. Static program point.

The call supplies a program segment track number and a track relative address. In the call the full program point is formed and put into the formal location. This is used for actual parameters which are compound expressions and for the return point. The codes representing compound expressions (thunks, see ref. 11) have been placed at the end of the call proper by the translator.

4. Simple variable.

The call supplies the DISPLAY reference and relative address of the variable. The call calculates the absolute address and puts it in the formal location.

The procedure call subroutine will transform the descriptions of the actual parameters and transmit them to the stack one by one. In doing so it need not refer to the procedure to be called. The procedure body will be able to identify each formal location by using the top of the stack as

reference point. In fact, since our stack starts at the high address end of the store the description of the return point will be found at the address "last used in stack" while the description of actual parameter no. p is found at the address "last used in stack" $+p$.

3.7. *Referring to formal parameters.*

The outstanding advantage of using explicit stored descriptions of the quantities of the program, including the program points, is that the translated ALGOL program need not distinguish between formal and non-formal versions of the quantities. Indeed, once a quantity description has been placed in a formal location the procedure body may refer to it exactly as though it had been non-formal.

The only formal parameters which receive a special treatment when referred to from the procedure body are those which are specified as **integer**, **real**, **Boolean**, or **label**. It may be noted that the reason for this anomaly is that we do not wish to treat simple non-formal variables through a general administration based on descriptions—this would entail an undue loss of storage space and execution speed. When referring to a simple formal parameter called by name a routine in the running system will examine the description of the actual parameter given in the formal location in the stack. Depending on the description the actual parameter is one of three things: (1) a constant, (2) a simple variable, or (3) an expression. The contents of the formal location in the three cases is: (1) the value of the constant, (2) the absolute address of the variable, and (3) the description of the program point where the actual parameter expression code (thunk) starts. The routine is expected to place the address of the actual paramter in the so-called universal address. This is simple in cases (1) and (2). In case (3) the routine must form the complete program point description of the point following the reference to the formal and place it at the top of the stack and transfer control to the point described in the formal location, the thunk. The code representing the thunk will then perform its task and will place the appropriate address in the universal address and will finally return to the point described at the top of the stack.

3.8. *Arithmetic expressions and subscripts.*

The running system is rather simple-minded about arithmetic expressions. All arithmetic variables are represented as floating point numbers within the machine. The difference between integer and real types only makes itself felt in round-off operations.

Likewise subscripting is treated by the most straightforward method: the machine code evaluates the subscripts one by one and forms the product sum of these values and the coefficients kept in the dope vector, using floating point operations throughout. A 9-word subroutine in the running system then calculates the final machine address and checks that it lies within the proper bounds.

3.9. *Storage of program.*

In trying to automize the transfers of program segments from the drum to the core store it is tempting to make use of the segmentation of the program which is defined in the block structure. The block structure was in fact used in this manner in DASK ALGOL (Ref. 4). However, we have since then realized that this approach has serious disadvantages. The point is that a block by definition is a unit with respect to the scope of its identifiers. However, what we need for segmentation is the dynamic units of the program. These are rather difficult to detect at translate time. In simple cases the most important dynamic units are small loops comprising much less than a block. In fact, the first part of a block is usually just intialization which is devoid of interest as soon as the real work begins. A more complicated situation is shown in the following fragment of a program which will be a dynamic unit while the solution of the equation

$$z \uparrow 2 + (1 - (z+1) \times \ln((z+1)/z) \times y = 0$$

is being found by bisections:

```
...
procedure Bisection(F,x,...); real F,x; ...
    begin ...
    for q := q/2 while abs(q) > eps do
        x := (if F > 0 then q else -q)+x
    end;
...
Bisection(z ↑ 2+(1-(z+1)×ln((z+1)/z)×y,z,...)
...
```

For each cycle of the for statement the control will pass from Bisection to the expression in the call, from there to the ln-routine, then back to the call and finally back to Bisection. This means that the most important dynamic unit is composed of three pieces of program taken from widely seprated places in the text.

On the basis of considerations of this kind and inspired by the scheme

adopted for the Ferranti Atlas (Ref. 8) we decided on the following scheme for the storage of the programs within GIER ALGOL:

1. No part of the instructions representing the algorithms of the program will be held permanently in the core store.

2. The program of instructions is held on the drum and is segmented by the compiler into the drum tracks so as to waste no space on the tracks.

3. The program segment stored on a drum track is arranged in such a way that while it is executed it may be stored in any place in the core store and will make no assumptions as to other program segments being present in the core store.

4. At any time the part of the core store which is not reserved for variables will be available for as many program segments as it will hold. The number of segment places will be allowed to vary between 2 and 20.

5. Program segments are transferred from the drum to the core store when the execution of the program requires them.

6. The running system keeps a table of the program segments currently held in the core store. When the program calls for a transfer of control to another segment the running system will try to avoid transferring the segment from the drum by first searching through this table.

7. When the program calls for a transfer of control to a segment which is not in the core store and there are no more unused segment places left in the core store, the segment will be overwritten on that segment held in the core store which for the longest time has been left unused.

This scheme could be realized quite conveniently in GIER. In particular the availablity of addressing relative to the order-counter could be used for references within the same segment. This is used, not only for jump instructions, but also for references to constants. In fact, as one unusual consequence of this scheme the literal constants of a program are never compiled into a table common to the program, but only into tables which are separate for each segment. In this way the constants used from a segment will follow this segment and not make any permanent demands on the core store. This requires that constants are taken through all stages of the translation as a unique class of objects.

Further details of this scheme are given in the following sections. In the machine code of the running system the corresponding algorithms require about 80 words.

3.10. *Program segment priorities.*

The rule of point 7 above is applied with the aid of a priority associated with each available segment. The running system keeps a current

priority, which is the priority of the segment which is currently in action. Each time the control is transferred to a segment the current priority is increased by one and is also assigned as the new priority of the segment. In this way the difference between the current priority and the priority associated with a segment in the core store shows how many segment transfers have been made since the segment was last used. In particular the segment having the lowest priority will be the one which has been left unused for the longest time.

Additional complications arise because in general the number of available segment places in the core store will vary. Clearly the system should in general be prepared to use all the available core store space for program segments. However, it is desirable that the reaction of the system in the case of an increase of the amount of storage available for segments should be somewhat cautious because the increase might be due to only a transient cancellation of reservations in the stack. For this reason the following procedure has been adopted. Irrespective of the amount of free store, new segment places are only put into circulation one at a time and only at the time when an actual transfer of a segment from the drum is taking place. Further this new segment place is not used for the segment actually being transferred, but is left empty with the priority one less than the new current priority. An additional advantage of this method is that the actual process of initializing a new segment place may use the drum transfer time which would otherwise be wasted.

3.11. *Example of segment allocation.*

As an illustration, let a program cycle indefinitely in a simple loop consisting of the segments stored on the tracks 91, 92, 93, and 94. The following table describes the development of the situation from the initial state, set by the translator, until all four segments are present in the core store. Each line corresponds to a situation. The columns

	Segment place												
	1		2		3		4		5		6		Current priority
Initial state	—	−1	—	0									0
After entry into 91	91	1	—	0	—	0							1
– – – 92	91	1	—	0	92	2	—	1					2
– – – 93	91	1	93	3	92	2	—	1	—	2			3
– – – 94	91	1	93	3	92	2	94	4	—	2	—	3	4
– – – 91	91	5	93	3	92	2	94	4	—	2	—	3	5
– – – 92	91	5	93	3	92	6	94	4	—	2	—	3	6

give, for each segment place, the drum track number of the segment stored in that place and the priority of that segment. If no track number and priority is given it means that the segment place is not regarded as available. The current priority is given in an addtional column.

If the loop had comprised 5 segments the 5th would have been put into segment place 1, thus overwriting segment 91. This indicates the tendency of the scheme to keep the program at the lower segment places. This tendency is further enhanced by the final process involved in the program allocation, the cancellation of segment places on priority overflow described in the following section.

3.12. *Program storage clean-up.*

Priorities will always be kept in the range 0-511. Whenever the priority counter reaches 512 the opportunity is taken to perform the following clean-up of the situation: priorities from 0 to 255 are replaced by 0, those from 256 to 511 by 1, while the track which is just being entered will be assigned the priority 2. Moreover the track places are checked in sequence, beginning with the one closest to the stack of variables, and all those of new priority 0 are removed from the list until the first one having the priority 1 or 2 is found or there are only two segment places left. The principal aim of this clean-up is to cancel those segment places which are never used. This may be of importance in short programs, because it will reduce the number of unsuccesful tests for coincidence of the new track number and the track numbers kept in the segment table performed at each transfer of control to a segment.

REFERENCES

1. Krarup, T., and Svejgaard, B., *GIER, Logical Organization.* Ingenioren, International edition, vol. 5, Dec. 1961, no. 4.
2. Naur, P. (ed.), *A Manual of GIER ALGOL.* Regnecentralen, Copenhagen 1963.
3. Jensen, J., and Naur, P., *An Implementation of ALGOL 60 Procedures,* BIT 1 (1961), 38.
4. Jensen, J., Mondrup, P., and Naur, P., *A Storage Allocation Scheme for ALGOL 60.* BIT 1 (1961), 89; Comm. ACM 4 (Oct. 1961), 441–445.
5. Jensen, J., Jensen, T., Mondrup, P., and Naur P., *A Manual of the DASK ALGOL Language.* Regnecentralen, Copenhagen, 1961.
6. Irons, Edgar T., *A Syntax Directed Compiler for ALGOL 60.* Comm. ACM 4 (Jan. (1961), 51–55.
7. Dijkstra, E. W., *ALGOL-60 Translation.* ALGOL Bulletin Supplement no. 10, Math. Centrum Amsterdam, Nov. 1961; Annual Review of Automatic Programming Vol. III, 327–356. Pergamon Press, London, 1963.

8. Fotheringham, John, *Dynamic Storage Allocation in the Atla, Computer, including an Automatic Use of a Backing Store.* Comm. ACM 4 (Oct. 1961), 435–436.
9. Samelson, K. and Bauer, F. L., *Sequential Formula Translation.* Comm. ACM 3 (Feb. 1960), 76–83.
10. Dijkstra, E. W., *Recursive Programming.* Num. Math. 2 (1960), 312–318.
11. Ingerman, P. Z., *Thunks.* Comm. ACM 4 (Jan. 1961), 55–58.
12. Grau, A. A., *The Structure of an ALGOL Translator*, Oak Ridge National Laboratory report ORNL-3054, 1961.
13. Arden, B. W., Graham, R. M., *On GAT and the Construction of Translators.* Comm. ACM 2 (1959) no. 7, 24–26.