THE WHETSTONE KDF9 ALGOL TRANSLATOR

B. RANDELL

The English Electric Company Ltd., Atomic Power Division, Whetstone, England

1. Introduction

Past experience with computers and translation schemes at the Atomic Power Division has shown that users' requirements of an automatic programming scheme are to some extent conflicting. On the one hand the price paid for ease of writing and testing in a convenient language must be small, and, particularly for large or frequently used programs, the final running efficiency must be high. The first objective requires extremely fast translation the second requires a large sophisticated translator with consequent increase in translation time. Possible solutions to this problem are, firstly, to have a compromise scheme, perhaps capable of varying degrees of sophistication or secondly to satisfy the two requirements with two translators.

The second solution has much to recommend it, and has indeed been chosen. A team at the Data Processing Division is working on a large multi-pass optimizing translator, with the aim of producing as efficient a running program possible. In parallel a smaller translator is being developed at Whetstone, due to be ready as quickly as possible after completion of the prototype KDF9, in which great stress has been laid on speed of translation, and on ease of communication between object program and programmer. This separation translators enables each scheme to pursue its own objectives to the full, and allows the writers of the large scheme more time to pursue their goal of run time efficiency.

Naturally the two schemes are to have complete compatibility, and should enable a user to test a new program quickly and efficiently, and then recompile it to get an efficient running program. Both schemes will accept full ALGOL 60, with only the following restrictions: 1. no dynamic own arrays; 2. no integer labels; 3. obligatory specifications for all parameters.

The Whetstone translator takes advantage of the fact that due to the high speed of operation of the KDF9 computer, there is considerable scope for utilizing the time spent during input of a program. This leads naturally the development of a scheme for strict one-pass translation of an ALGOL program, so that the object program is complete, ready to be obeyed, almost immediately after the reading of the ALGOL ceases.

2. The KDF9 Computer

Very little mention need be made of the KDF9 computer on which this translation scheme is being implemented. The basic machine configuration needed is an 8k core store, paper tape input/output, monitor typewriter, and two magnetic tape units. Each word of core storage is of 48 bits, made up of six 8-bit syllables. Only

within the translation routine and the subroutines that make up the run time control routine is full use made of the rather novel nesting-store accumulator, which thus need not be considered in this paper.
Although the KDF9 is a binary machine the normal method of preparing machine code programs will be in User Code, an alphanumeric assembly language.
For a more complete description of the KDF9 and its User Code, see references 1 and 5.

3. The Object Program

At an early stage in the project, the author and a colleague, Mr L.J. Russell, met Dr Dijkstra, of the Mathematical Centre, Amsterdam, who, with Mr Zonneveld, was joint author of the extremely successful ALGOL translator for the Centre's X1 computer. Later, at the invitation of Dr Dijkstra, a most pleasant week was spent at the Mathematical Centre discussing translation techniques (Randell and Russell, 1962).

The object program generated by the Whetstone translator is a development of that of the X1 translator. Thus the object program consists of a set of operations, with parameters where necessary, which use the remaining core storage as a stack (Dijkstra, 1960) containing all the currently available declared quantities. The stack system is a natural consequence both of the block structure of ALGOL and of the fact that recursive calls can be made on procedures. The object program is obeyed interpretively at run time by a control routine, which performs the normal arithmetic and logical operations, dynamic storage allocation of arrays, and all the necessary stack manipulation operations.

The requirements of parallel declarations and one-pass translation have their effect on the design of the object program, which must be capable of being partially generated as the ALGOL is read in, and then completed when any previously missing declarations are found.

1. Arithmetic Expressions

The method of using a stack for the evaluation of an arithmetic expression is quite straightforward, and has been described by Dijkstra (196Ia). Essentially any necessary operands are copied from their positions in the working storage section of the stack, into the top position of the stack, with information as to type, etc. All arithmetic operations work on the top one or two stack positions. Stack positions which contain a value, an address or a label, and information pertaining to the quantity, occupy a double-word of storage, called an accumulator, whereas the type and meaning of any quantity in working storage is always known at translation time, and can thus be given implicitly in any operations referring to it. The precedence of operations implied in an arithmetic expression is given explicitly by a re-ordering of operations into 'Reverse Polish' in the object program.

Example

$x := i + y;$ (x, y of type **real**, i of type **integer**)

This is re-ordered into the Reverse Polish form

x, i, y, +, :=

This bears a close resemblance to the actual object program which is generated. In fact, in the object program the above identifiers are represented by 'Take Result' or 'Take Address' operations, each with a parameter used to find the location assigned to the identifier.

Thus the above statement is translated into

TRA	*x*	(Take Real Address)
TIR	*i*	(Take Integer Result)
TRR	*y*	(Take Real Result)
+		(Add)
ST		(Store)

(The identifiers *x*, *i*, *y*, here represent stack addresses).

The extent of the stack is always given by a counter *AP*, the Accumulator Pointer, counting in units of one word. *AP* always indicates the first free store after the end of the stack, i.e. the store into which the next quantity to be stacked would be placed.

Then in the above example, if *AP* is originally, say 12, the detailed action of the operation is:

TRA	*x*	Address of *x* set in Stack [12] and a bit pattern meaning 'real address' in Stack [13]. Two added to *AP*.
TIR	*i*	Value of *i* set in Stack [14], 'integer result' in Stack [15], two added to *AP*.
TRR	*y*	Value of y set in Stack [16], 'real result' in Stack [17], two added to *AP*.
+		Examines positions *AP* - 3, *AP* -1 (i.e. 15, 17) does the necessary conversion of *i* in Stack [14], sets the result *i* + y in Stack [14], 'real result' in Stack [15] and decreases *AP* by two.
ST		Works on top accumulator (a result) and the next accumulator (an address) does any necessary type conversions, performs the storage, and decreases *AP* by four.

2. Procedures and Blocks

At any point during the execution of the object program the stack will contain for each procedure: 1. an accumulator space for a resulting procedure value; 2. link data; 3. parameters; 4. declared scalars, labels, array words; 5. array storage; and 6. anonymous intermediate results. Apart from the procedure value accumulator, and the parameters, a block is treated as a procedure, and in what follows the two terms are used interchangeably.

Thus a recursive activation of a procedure will be shown by more than one such set of stacked information. The link data performs a double purpose to provide access to the declared variables in the last activation of every procedure, and to provide a means of unravelling a set of nested procedure calls, through

every activation of each procedure. Declared variables in stacked working storage are addressed relative to the starting position of the stack information pertaining to the block in which they are declared (this position is called *PP* - the procedure pointer). Since the amount of link data is fixed, the addresses, relative to *PP*, of parameters and declared variables can be assigned at translation time. Thus the first function of the link data (providing access to declared variables) requires a scheme for finding the value of *PP* for the last activation of each procedure, which can then be used with the relative address, to form the actual stack address of a declared variable. The second function of the link data is performed quite simply by maintaining a list of successive values of *PP*, for each activation of each procedure. (This list is called the dynamic chain, whilst the list of *PP*'s for the last activation of each procedure is called the static chain.) At any stage only the current procedure, and its surrounding procedures, are of any interest. A Block Number (*BN*), starting at one for the main program level, is allocated for each lexicographic level of block or procedure. Thus the static chain has one value of *PP* for each level of the program, up to the current level. For the sake of efficiency the static chain is duplicated in a vector called *DISPLAY*. Then reference to any declared variable is by means of a dynamic address (*n*, *p*), where *n* is the lexicographic level of the block in which the declaration occurs, and *p* is the stack address of the variable relative to the *PP* of the block. The dynamic address is evaluated as '*DISPLAY* [*n*] + *p*'. It would be possible to find the appropriate *PP* without using *DISPLAY*, by just working back down the static chain in the sets of stacked link data until the right level was encountered, but this is obviously inefficient.

The dynamic chain is rather more complicated as it is this mechanism that allows the stack to be restored to its original condition after exit from a procedure. Taking an example of a block *a* containing a procedure call on a procedure *b* declared in a block *c*.

When the call is made the next available accumulator (indicated by *AP*) is left for a possible result from the procedure and after completion of the entry to the procedure the stack is:

	Core No.		
	0 1	}	Procedure Value Accumulator
APa, PPb:	2	*PPc*	Static Chain
		PPa	Dynamic Chain
	3	*BNb, WPb*	Block Number, Working Storage Pointer
		FP	Formal Pointer
	4	*RAa*	Return Address
		LINK	User Code Link
	5 6	}	1st Formal Parameter
	7 8	}	2nd Formal Parameter
	9		1st Working Store
	10		2nd Working Store

11 3rd Working Store
WPb, APb: 12

In this example of a procedure with two parameters and three working stores, core numbers have been given arbitrarily from zero.

The ruling value of *PP* after entry to procedure *b* indicates the start of the link data, and is equal to the value of *AP* before entering procedure *b* from procedure *a*. The dynamic chain, in the second half of core 2, is the value of *PP* ruling during procedure *a*. The block number and amount of working space of the procedure *b* are known at translation time, and are given as parameters to the object program operation which performs the entry to the procedure for storage in the link data. The static chain is the *PP* of the block containing the procedure *b*, i.e. of the one with a block number one less than that of *b*. This could be found by working down through the chained link data, but in fact can be obtained directly from the vector *DISPLAY*. The new value of *PP* is added to *DISPLAY* at the position given by the block number of *b*. The formal pointer is originally set to indicate the first formal parameter accumulator, and is used in processing the parameters at a procedure call. The remaining link data consists of a return address and a User Code link. When a procedure is left the return address is used to reset the object program counter. The User Code link indicates the place within the control routine from where a procedure was called. Control is returned to this point, to finish any necessary functions of the control routine before calling for the next object program operation (as given by the reset object program counter).

This system for using a stack for procedures, possibly recursive, is fairly straightforward - the main complication is in ensuring that *DISPLAY* always contains a copy of the static chain. Whenever the validity of *DISPLAY* is in doubt it must be checked against the actual static chain given in the sets of stacked link data. This must be done on all formal procedure calls, and on leaving any procedure, normally or by a go to statement.

Example

```
begin  integer i;
       procedure Q;
Q1 :   begin  real x;
              . . . . .
       Q2:    begin  real y;
                     . . . . .
                     y: = 0;
                     . . . . .
              end;
              . . . . .
       end;
       . . . . .
P1:    begin  real a;
              . . . . .
       P2:    begin  real b;
```

```
                    . . . . .
          P3:       begin  real C;
                           . . . . .
                           C: = 0; Q
                    end
               end
          end
     end
```

In this example, where dots indicate further statements, indentation has been used to show the various levels. Each block has been labelled, so as to indicate the level at which the label occurs, and the fact of being part of procedure *Q* or of the main program. It should be noted that a procedure whose body is itself a block only causes a single change in level.

The system of dynamic and static chaining, and the associated vector *DISPLAY* are illustrated using the above labels. For this purpose the main program is assumed to have the label *PROGRAM*.

i. At the assignment of zero to *C* only the blocks labelled *PROGRAM, PI, P2* and *P3* have been activated. The static and dynamic chains coincide, chaining together the four sets of link data in the stack, which correspond to the above four activations of blocks. Four entries have been made in display.

DISPLAY [1] = *PROGRAM*
DISPLAY [2] = *P1*
DISPLAY [3] = *P2*
DISPLAY [4] = *P3*

Here the labels in fact indicate stack addresses - the values of *PP* for each activation of each block.

ii. However, at the assignment of zero to *y*, in the call of procedure *Q* in block *P3*, the situation is more complicated. Six blocks have been activated, but only three (*PROGRAM, Q1* and *Q2*) are currently valid, in the sense of having declarations which could pertain to the situation at the statement '*y* : = *0*'. Thus the stack must still contain activations of blocks *P1, P2* and *P3*, hidden under the activations of blocks *Q1* and *Q2*.

As a result the dynamic chain links together, in order,

PROGRAM, PI, P2, P3, Q1 and *Q2*

(this list, read from right to left, gives the complete set of blocks to be unravelled by working out through the various **end**'s),

whilst the static chain only links together

PROGRAM, Q1 and *Q2*

Therefore at this time *DISPLAY*, which must mirror this static chain contains only three entries

DISPLAY [1] = *PROGRAM*
DISPLAY [2] = *Q1*

DISPLAY [3] = *Q2*

3. Parameters

On entry to a procedure the parameter operations in the object progran are processed to set up the formal parameter accumulators in the stack following the link data. For instance an actual parameter which is just a real variable is translated into a parameter operation with its dynamic address. At the procedure call this causes the corresponding formal parameter accumulator to be set up with the evaluated dynamic address together with an identifying bit pattern. The evaluation of the dynamic address is done before any adjustments to *DISPLAY*.

Of more interest is an expression as actual parameter corresponding to a formal parameter called by name. An implicit subroutine of object program operations is generated, with a parameter operation giving the address of this subroutine. At a procedure call this parameter operation causes the formal parameter accumulator to be set up with the address of the subroutine, and the *PP* ruling at the time of the procedure call.

A call on this parameter causes the subroutine to be entered in much the same way as a procedure is entered, stacking vital information such as *PP, AP*, etc., during operation of the subroutine. Before completing entry to the subroutine, the value of *PP* in the formal parameter accumulator is used to set up *DISPLAY* in the state it had at the procedure call – this enables correct evaluation of the expression, as it is written in terms of declarations valid at procedure call. After the subroutine has completed the evaluation of the expression the normal mechanism for leaving a procedure resets conditions to those pertaining before entering the subroutine, but with the result of the expression in the top accumulator, just as if it had been placed there by a 'Take Result' operation.

4. Arrays

The organization of arrays in the Whetstone translator is fairly conventional. A word is set aside in working storage for each array, to contain the base address of the array, the starting address of the array, and the address of the storage mapping function of the relevant array segment. After entry to a block, operations are obeyed for each array segment, to set up the mapping function (the coefficients of a polynomial for evaluating the address of a subscripted variable), the array words, and space for the arrays, and to increase the working space, as indicated by the stacked working space pointer, to include the mapping function and the arrays.

Arrays as formal parameters are dealt with by handing on the address of the relevant array word. In the case of an array called by value the array is copied into the stack, after the current working space, and a suitably adjusted copy of the array word set up in the formal parameter accumulator.

5. Labels and Switches

Labels are allocated space in the working storage of their block. This space is set up with the appropriate value of the object program counter, and the current value of *PP*. These labels can then be manipulated in designational expressions,

and handed on, perhaps repeatedly, as formal parameters, the value of *PP* being used if necessary to adjust *DISPLAY* when the label is actually used.

A switch can be thought of as a 'label procedure', which delivers the result of one of a set of designational expressions. Entry to a switch is like entry to a procedure, and thus can go recursive in a similar fashion. The end product is a label in the top accumulator, ready for the operation generated from the go to statement.

6. For Statements

A for statement is organized by a portion of the control routine (the for routine) which uses sections of the object program, for evaluating the various arithmetic and Boolean expressions, and the address of the controlled variable, as subroutines. This is done by stacking a User Code link to be found by a special object program operation at the end of each subroutine, which then returns to the appropriate point in the for routine. The system of making the expressions and controlled variable into subroutines allows each one to be translated once, in the order in which they appear, even for the step-until element, which has to simulate an expanded ALGOL version with repeated reference to these quantities.

A for statement is automatically made into a block, and thus sets up the normal stack positions for link data, etc. Then at any point where an interruption of the work of the for routine can occur, all relevant information is kept in the stack. Such an interruption will occur when the controlled statement involves a for statement (explicitly, or within some procedure call) or when evaluation of an expression in a for list element, or of the address of the controlled variable, calls on a function designator which involves a for statement.

4. The Translation Routine

The translation routine operates on the ALGOL program as it is read in, generating the object program in the core storage. The re-ordering of operations is performed using a translator stack and a system of priorities. This handles precedence of arithmetic operators, etc., and is used also for the bracket structure of ALGOL. A name list is used to contain details of identifiers encountered in the current and all the surrounding blocks. As there is no opportunity to scan the ALGOL program a set of state variables is used to differentiate between different uses of the same delimiter. The basic system is to read the ALGOL program until a delimiter is found, noting if an identifier or a constant has been read in. Each delimiter has its own routine, which has at its disposal a set of general subroutines for performing the various tasks common to two or more delimiters.

The need for strict one-pass translation necessitates being able to proceed with the generation of object program when lacking declarations of identifiers, and in fact never being able to use any non-local declarations until the end of a block is reached.

The arbitrary order of declarations can cause difficulties. For example, declarations for non-local variables to a procedure can appear after the procedure declaration. The context in which an identifier is used is not always helpful – in particular the case of use as an actual parameter to a procedure. The name list is

used to contain declaration information, if available, or the sum total of information garnered from the various uses of an identifier.

Being able to proceed with object program generation is facilitated by designing the object program so that the operations generated from a given use of an identifier take a similar form no matter what kind of declaration is subsequently found. For instance, what appears to be a subscripted variable in an actual parameter list may in fact turn out to be a switch designator.

1. The Translator Stack

By assigning priorities to certain of the ALGOL delimiters the translator stack can deal very simply with questions of arithmetic precedence, bracket structure, conditional statements, etc. Essentially the stack is used as a holding store, and is used to re-order the ALGOL into Reverse Polish, which is the form of the object program, as described in Section 3. The basic system can be illustrated on a simple arithmetic expression

$$A + B \times (C - D / (E + F \times G)) \uparrow H$$

This input string is read from left to right, and symbols can be transferred to the output string, or can go into the stack (used simply as a last in – first out store). Each operator has a priority; this is compared with the priority of the operator at the top of the stack, and determines the action to be taken. 1. Operands are transmitted straight to the output. 2. Left parenthesis is stacked with a priority zero. 3. Right parenthesis causes operators to move from stack to output until a left parenthesis is uncovered. This is then removed. 4. The remaining operators each have a priority. As each operator is met, its priority is checked against the priority at the top of the stack, and operators are moved from the stack to the output until an operator with a priority less than the priority of this current operator is encountered. The input operator is then itself placed in the stack.

The priorities are		
	+, –	2
	×, /	3
	↑	4

(Unary '+' and '-' are not dealt with in this simple system.) Right parenthesis is essentially controlling unloading of the stack with a priority of one, and hence must uncover its matching left parenthesis.

By this means the above expression would be re-ordered into

$$A, B, C, D, E, F, G, \times, +, /, -, H, \uparrow, \times, +$$

However, the system can be greatly extended by using a double priority system, to deal with conditional expressions and statements, statement brackets, etc. The double priority is necessary because certain delimiters perform what is essentially a double function. For example **then** can be used to terminate a Boolean expression and to precede a statement. In general each delimiter is given a 'stack priority' and a 'compare priority'. As their names suggest, the stack priority accompanies a delimiter when placed in the stack, and the compare priority is used to control unloading of the stack.

Thus **then** must unstack through a Boolean expression, which could be conditional, to its corresponding **if**, and **until** must unstack through an arithmetic expression to the delimiter **step**, etc.

Some delimiters, such as opening parentheses, **begin**, etc., go straight into the stack without doing any unstacking. Where necessary the translator stack is used to store away certain of the state variables with the current delimiter.

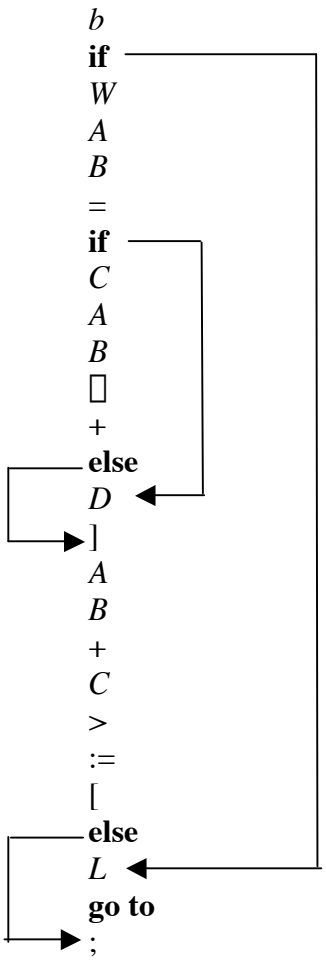
Example

The re-ordering performed using a stack and a double priority system can be illustrated on the following ALGOL statement.

if *b* **then** *W* [**if** *A* = *B* **then** *C* + *A* □ *B* **else** *D*] : = *A* + *B* > *C*

else go to *L*;

is re-ordered into



Here **else** indicates an unconditional jump, **if** indicates a jump conditional on the top accumulator being **false**, and ']' an indexing operation, which forms the address of a subscripted variable.

for resetting later. This avoids using explicitly recursive subroutines for dealing with, for instance, arithmetic expressions in a subscript in an arithmetic expression.

Syllable counters are stored with some delimiters, to permit later completion of the object program. For example at **then**, an incomplete implicit jump is generated, and the syllable number of its position stacked with the **then**. When the

corresponding **else** finds this in the stack the implicit jump around the expression or statement following the **then** can be completed.

It is necessary to differentiate between the **then** and **else** used in conditional expressions and in conditional statements, and this is done by using the state variables. Similarly the various uses of comma, for instance, in subscript lists, for clauses, switch lists, etc., must be differentiated.

2. The Name List

The name list is used to contain information about each identifier in the current block, and in all surrounding blocks. Thus the name list, like the translator stack, can fluctuate in size.

An entry is made in the name list at the declaration of an identifier or at the use of an identifier which does not appear in the part of the list pertaining to the current block. Thus entries consist of the actual identifier, a bit pattern specifying type or expected type, number of dimensions or subscripts if applicable, some markers concerned mainly with checking, and either the dynamic address which has been assigned to the identifier or chaining information. Chaining is used to link together incomplete object program operations which are generated for calls on an undeclared identifier. Because of parallel declarations and the system of chaining it is only ever necessary to search through the section of the name list pertaining to the current block, rather than through the complete list whenever an identifier is encountered.

3. The System of Chaining

All object program operations concerned with calls on identifiers consist of a one-syllable operation code and a two-syllable address. Only when an identifier appears as a 'declared' entry in the name list for the current block can a use of the identifier be translated fully. If a set of incomplete operations have to be chained together the two syllables left for the address are used to contain a link of the chain, and the single syllable for an indication of the type of use being made of the identifier. When the relevant declaration is found the chain is scanned, replacing the single syllable by the appropriate operation code, and the link by the address of the identifier. Each use of an identifier is checked against information gained from previous uses, which is kept in the name list. By the time the declaration is reached the name list will contain the 'logical sum' of all the information known about the identifier, and this is checked against the declaration information.

The chaining system is basically as follows.

i. At use of an identifier

The name list for the current block is searched. If it contains a 'declared' entry for this identifier then this is used, and no chaining is necessary.

If there is no entry a 'used' entry is added to the name list accompanied by two syllable counters, both set to indicate the position of the space left in the object program for the address of the identifier. These syllable counters, called *CS* and *CF*, indicate the start and finish of the chain for this identifier.

If there is already a 'used entry' the value of *CS* contained in it is placed in the object program in the space left for the address of this identifier, and then *CS* is replaced by the syllable counter of this address space.

ii. At declaration of an identifier

The name list for the current block is searched, and if there is no entry a normal 'declared' entry is added. If there is already a 'declared' entry, a failure indication is given.

If there is a 'used' entry the chain must be followed through to finish the various incomplete object program operations. This is done by starting at the position in the object program given by *CS*. Each address space contains the syllable counter of its successor, the end of the chain being marked by a blank address space. Finally the 'used' entry is replaced by a 'declared' entry.

iii. At the end of a block

At the end of a block the section of the name list for this block is collapsed by deleting any 'declared' entries, combining any 'used' entries with corresponding entries in the containing block, or adding 'used' entries on to the list of the containing block.

Combining entries will either cause a chain to be followed through, as described above, when the containing block contains a 'declared' entry, or will cause two chains of 'used' entries to be joined. This is done by putting the *CS* of the entry in the containing block in the address space indicated by the *CF* of the entry in the inner block, and replacing this *CS* by the *CS* from the inner block entry. The inner block entry for this identifier is then deleted.

This basic system is slightly complicated by the need to check that expressions in the bound pair list of array declarations contain no local identifiers.

4. State Variables

As has been mentioned above, state variables are used where necessary, to differentiate between the various possible uses of a delimiter and as counters, etc.

For instance the variable *V* is set to zero at **begin**, to one at declarations, and to two at statements. This is used to differentiate between blocks and compound statements, and also to check out the occurrence of declarations amongst statements.

A counter *L* is used for the number of declared identifiers assigned a working space position in the current block, and *NL* is set to indicate the start of the current section of name list.

At the beginning of a block, a procedure block, or a for statement block, *V, L* and *NL* are stacked with the **begin**. Then a new *L* can be started, *NL* can be set up for use inside the new block, and *V* zeroed, with the knowledge that translation can resume correctly for the outer block by unstacking the **begin** and restoring the *V*, *L* and *NL*, when the end of the new block reached.

Other important state variables are *E*, which is set to *0* for expressions, *1* for statements, and *TE* which is a bit pattern used in expressions to indicate the type of identifiers expected. *E* and *TE* are stacked with **if**, for example, allowing *E* to be zeroed and *TE* to be set to 'algebraic' (incorporates real, integer and Boolean)

for the Boolean expression that must follow. On reaching the corresponding **then**, the stacked value of *E* indicates whether it is to be take as **then***S* or **then***E* (conditional statement or conditional expression, respectively), and for **then***E*, *TE* indicates the type of expression in which the **if** occurred. Since types are determined, and checked, dynamically at run time, the translator does not perform a full check on compatibility of types Similarly checks on the compatibility of actual parameters with the specifications of their corresponding formal parameters are performed at run time.

Other state variables are used to mark progress through a for clause, to mark the fact of being within an actual parameter list, etc.

One important exception to the system of separate routines for delimiters is the translation of the heading of a procedure declaration. This uses no recursive definitions, and hence can be translated by its own special routine, which sets up parameter lists, information about value and specification parts, etc. Thus there need be no confusion between **real**, **array**, etc., used in declarations and as specifiers.

5. Use of Core Storage by the Translation Routine

During translation, space is required in the core storage for the translation routine itself, the object program being generated, the translator stack and name list, and a small fixed amount for the constants, state variables, etc.

Ignoring the fixed amount of storage required, space is thus needed for an object program which grows steadily in size, and for the name list and stack whose size varies continually, but ends up as zero. The system evolved is for the object program to start immediately after the translation routine, and for the name list to start at the other end of the storage, 'pulsating' towards the object program. The translator stack is placed in the centre of the store, to pulsate towards the name list. If the object program reaches the start of the stack or if the name list and stack reach each other the stack is moved, say 32 places, in the appropriate direction. If this would cause yet another clash translation is not possible and a failure will be indicated. This system allows maximum use of core storage, whether being used up by virtue of the size or of the complexity of the ALGOL program, and is illustrated in Fig. 1.

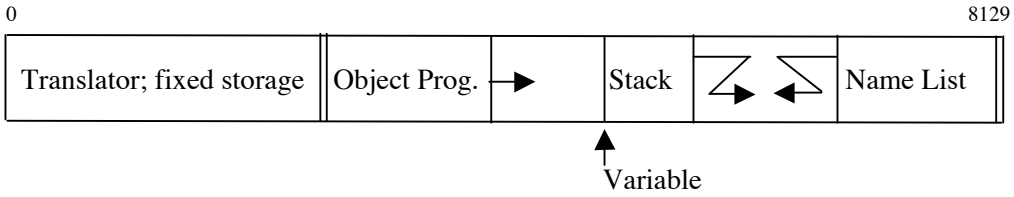


Fig. 1. Layout of storage during translation.

5. Code Procedures

A scheme for including procedures whose bodies are in User Code in an ALGOL program has been devised, and is to be implemented on both translators for the KDF9. Briefly this allows a normal procedure heading, with only the specifiers **procedure**, <type> **procedure** and **switch** being excluded, and then a

procedure body in User Code bracketed by the symbols **KDF9** and **ALGOL**. Within the User Code of the procedure body access is allowed to the parameters by permitting pseudo-instructions calling on the formal parameters. All other communication with the surrounding ALGOL program is expressly disallowed. The rules for the inclusion of User Code procedures are in fact the required expansion of section 4.7.8 of the ALGOL Report (Ref. 6).

The scheme is quite powerful, allowing for instance, calIs by name on arithmetic expressions, the use of Jensen's device (Dijkstra, 1961b), etc. The main reasons for designing such a scheme are to ensure compatibility between the two translators even for programs including code procedures, and to minimize duplication of effort between the two translator projects in providing input/output procedures, etc. Naturally casual use of this facility would impair ease of communication of programs and will be discouraged.

The implementation of procedures in User Code on the Whetstone translator has produced some interesting problems. In particular the method of communication at run time between an interpreted object program and a User Code procedure is of some interest but will not be discussed here, as being outside the scope of this paper.

6. Communication between ALGOL Program and Programmer

An important feature of the Whetstone ALGOL translator is the stress placed on ease of communication between ALGOL program and programmer. Thus it should be possible to check-out and test an ALGOL program with no knowledge of the generated object program.

The first problem is that of indicating to a programmer the point at which an error in his ALGOL program has become apparent to the translator. This is done by printing out the next 100 characters, or two lines, of the ALGOL (counting only non-trivial information). The line number, and the last label and last procedure identifier declared will also aid identification of the offending ALGOL. This information will be accompanied by a message describing the error (or rather, inconsistency).

It will be possible to continue, and search for further errors, only on the basis of making some reasonable estimate as to the cause of the inconsistency.

Error print-outs at run time will arise from two causes. Firstly, further ' checks on the validity of the ALGOL which are not easily checked at translation time are performed at run time (for example compatibility of formal and actual parameters). Secondly, actual errors in the program can result in numbers becoming invalid, subscript bounds being exceeded, etc. Error print-outs are formed using a 'failure tape' which has been set up at translation time. As any section of the name list is collapsed (see Section 4) a copy of the section is made on magnetic tape. This will include details of the block being collapsed (type of block, level number, line number at start of block, etc.) and information about every identifier in the block. The failure tape also includes, for each block, a table giving details of line number against object program counter. Thus using this failure tape, error print-outs can be made giving a description of the failure, the

position where it occurred, and details of the current block. This tape could also be used for trace routines, post-mortems, etc.

7. Acknowledgements

A large measure of any credit for the Whetstone ALGOL Translator must go to Mr L. J. Russell, who with the author, is jointly responsible for the design and implementation of the translator, and who has assisted in the preparation of this paper. The author's indebtedness to Dr E. W. Dijkstra, and also Professor A. van Wijngaarden, and Mr J. A. Zonneveld, will be obvious to anyone familiar with the ALGOL Translator for the X1 computer at the Mathematical Centre, Amsterdam. The author also wishes to thank The English Electric Company Limited for permission to publish this paper.

REFERENCES

1. G. M. DAVIS (1960). The English Electric KDF9 Computer System. *Computer Bull.* **4**, 119-120.
2. E. W. DIJKSTRA (1960). Recursive Programming. *Numerische Mathematik* **2**, 312-318.
3. E. W. DIJKSTRA (1961a). An ALGOL 60 Translator for the X1. *ALGOL Bull. Suppl.* **10** (Translated by M. Woodger from the German in MTW **2**, 54-56 and MTW **3**,115-119).
4. E. W. DIJKSTRA (1961b). Defense of ALGOL 60. *Comm. A.C.M.* **4**, 502-503.
5. English Electric (Data Processing and Control Systems Division) publication. KDF9 Programming Manual.
6. P. NAUR (1960). Report on the Algorithmic Language ALGOL 60. *Comm. A.C.M.* **3**, 299-314, and *Numerische Mathematik* **2**, 106-136.
7. B. RANDELL and L. J. RUSSELL (1962). Discussions on ALGOL Translation at Mathematisch Centrum. English Electric Report W/ AT 841.