Form F. 1624.

V39-39

THE ENGLISH ELECTRIC COMPANY LIMITED

DEPT.: Atomic Power Division.

WHETSTONE.

REPORT No. W/AT 841
A.C.E.D. Ref. No.
Deptl. Ref.
Date

Copies	TO:—	
	Mr. E. M. Price	
	Dr. H. S. Arms	
	Mr. P. H. W. Wolff	
	Mr. J. D. McKean	
	Mr. A. J. Joyce	
	Dr. D. M. Parkyn	
	Mr. B. Randell	
	Mr. L. J. Russell	
	Library	
	Prof. Dr. Ir. A. v. Wijngaarden	Mathematisch Centrum, Amsterdam
	Dr. E. W. Dijkstra	" " "
	Mr. W. E. Scott	Kidsgrove
	Mr. C. Robinson	"
	Mr. F. G. Duncan	"

Discussions on Algol Translation, at Mathematisch Centrum.

Report by
B. Randell
L. J. Russell

Summary

This report is a record of discussions with Dr. E. W. Dijkstra, at Mathematisch Centrum, Amsterdam, during 4th - 8th December, 1961.

Topics covered included details of the Algol Translator written at Mathematisch Centrum, and proposals regarding the techniques to be used in the Algol Translator being developed at the Atomic Power Division, Whetstone.

Authorised by: Dr. D. M. Parkyn

Index

1. Introduction

This report is a precis of the notes brought back from the visit of the authors to Dr. E. W. Dijkstra, of Mathematisch Centrum, Amsterdam.

During the visit Dr. Dijkstra described the logic of the original Algol translator for the XI computer, written by himself and Dr. Zonneveld, and, where applicable, details of the latest Algol translator now being prepared for the XI at Mathematisch Centrum. Naturally, working experience of a translator has proved very valuable, and as a result several improvements as the original translation methods have been developed. Considerable time was spent on discussing requirements for the KDF9, and a large measure of agreement was reached on translation techniques. This report is intended to be a complete record of the Amsterdam discussions, and hence includes propesals which have since been modified.

1.1 General Outline

The result of the week's discussions was a decision by the authors to produce an Algol Compiler according to the following specifications.

a) One pass load-and-go translation into an object program

b) The object program (developed from that used in XI Algol) to be obeyed interpretively.

c) The input language will be ALGOL 60, with the following main restrictions.

   i) No dynamic own arrays

   ii) No integer labels

   iii) Obligatory specification of all parameters

d) Great attention to be paid to checking, both at translation and at run time.

2. The Object Program

The object program produced by the KDF9 Algol compiler will be a development of the form used in the XI compiler. Thus it is based on the principle of a stack, necessary for the complete recursiveness allowed in Algol.

The stack will contain the declared variables for each incarnation of a procedure (we include block as a special form of a procedure and use the two terms interchangeably). Thus a variable is referred to by giving its position relative to the stack position at the start of the current entry to the block in which it was declared.

The object program will be obeyed interpretively, causing entry to be made to various parts of a sub-routine complex. It is only within the sub-routines of the complex that an attempt will be made at efficient use of the KDF9 order code.

Integer, real, and Boolean variables will each use one word of storage. Anonymous intermediate results in the stack will be accompanied by a second word, giving information as to type etc.

2.1 Arithmetic

Arithmetic expressions will be translated into a form of Reverse Polish, obeyed interpretively using a stack technique. The normal singlelength working of the KDF9 will be used. The KDF9 nesting store, however, will be used only within the sub-routine complex.

All declared variables, and constants will be one word in length. Information as to their type will be carried in the operations referring to them. Anonymous intermediate results will carry a second word with them, giving, among other things, type information.

Types integer and real will naturally correspond to KDF9 fixed and floating-point. All operations will be checked, and it is proposed that the device 'semi-real' be used to avoid integer overflow. A semi-real number is one which, according to declarations, should be of type integer, but which has become invalid in fixed-point notation, and hence is now represented as though it were from a real declaration. A failure occurs if an attempt is made to assign a semi-real number to an integer, or to compare integer & semi-real numbers in a relation. An operation between integer and semi-real will be of type semi-real, and between semi-real and real will be of type real.

As regards floating point overflow, it is suggested that out of range numbers be replaced by a correctly-signed 'machine infinity'. This is a floating point number, the theoretical maximum of the KDF9 number representation.

2.1.1 Exponentiation

Because numbers carry their type with them, exponentiation can be done as in the Algol report. In the case of a positive integer exponent, the repeated multiplication is done by shifting down the exponent and checking its lowest digit, so that after squaring the base one may or may not multiply the partial result by it. Multiplication uses the routine in the sub-routine complex, which takes care of types.

2.2 Constants

Constants are stored within the object program, taking up one word. A constant is stored each time it is used. Type information is implicit in the operations calling on these constants.

2.3 Arrays

Arrays are stored as a set of consecutive storage locations. Information given by the declaration, used to address individual elements, is given in the storage mapping function.

An n-dimensional array declaration is translated into 2n expressions, followed by an operation. These expressions are lower and upper bounds for each dimension. The operation processes these expressions to form the storage mapping function.

Thus the physical address of $A\left[S_0, S_1, \ldots S_{n-1}\right]$

is $\alpha_0 + \sum_{j=0}^{j=n-1} S_j \Delta_j$

α_0 is address of $A[0,0, \ldots 0]$, whether within the array or not

$\Delta_0 = 1$ as each element takes 1 word

$\Delta_{j+1} = \Delta_j \times$ (upper bound $_j$ - lower bound $_j$ + 1)

Δ_n, by means of this recurrence relation, is the size of the array.

We also need the starting address, for checking purposes.

So ignoring Δ_o, the storage mapping function has n + 2 items, namely

α_o starting address, $\Delta_1, \ldots \Delta_n$.

2.3.1 Multiple Array Declarations

In the XI compiler arrays with the same bounds have separate storage mapping functions.

Taking an example

<u>array</u> A, B $[l_o:u_o, l_1:u_1]$

This gives rise to two storage functions, whose length is not known until the closing bracket is found. The two functions are to be set up after entry to the block in first order storage.

```
The object program generated is TAKE  l_o
                                TAKE  u_o
                                TAKE  l_1
                                TAKE  u_1
                                Specify Local Variable Counter
                                Specify No. of Arrays (2)
                                Make Storage Function
```

The number of dimensions can be found by the operation 'Make Storage Function', from the effects on the stack of the evaluation of the expressions. The storage function is calculated, and then set up for each array, with adjustments to array start positions, and with type information. Then WP - a counter denoting the beginning of working space in the stack, is increased past the space needed for the arrays.

The mapping functions are put into the stack under control of the Local Variable Counter (L.V.C.)

At translation time the array identifiers had been stacked, until the set of bounds were finished, and had been emptied into the name list.

Thus in this case entries are,

B, block no, L.V.C.
A, block no, L.V.C. + (n + 2)

and the translator increases L.V.C. by 2 x (n + 2)

2.3.2 Subscripted Variables

A subscripted variable is represented in the object program by the address of its mapping function, then a set of arithmetic expressions, followed by the operation Index

Address (INDA), or Index Result (INDR), which has with it the number of subscripts.

The result in the stack, on entry to operation INDA, is an address, hidden by a set of real numbers and integers, the results of the evaluation of the arithmetic expressions. The operation INDA works through the stack, doing any necessary conversions to type <u>integer</u>, until it finds an address. The number of subscripts is checked and the mapping function used to evaluate the address of the subscripted variable. This address is left in the stack.

Operation INDR is similar, but leaves the value of the variable in the stack.

2.4 Assignments

A simple assignment statement, say X: = EXP, becomes in the object program an operation for finding the address of X, then program for evaluating the expression, followed by a 'store' operation.

There are three cases to be considered.

a) X a declared variable (or formal parameter called by value)

b) X a formal parameter called by name

c) X a procedure identifier

2.4.1 Non-Formal Assignments

The above example becomes in the object program

```
Take Real Address  )              ( Take Integer Address
Take Expression    )      or      ( Take Expression
Store              )              ( Store
```

depending on the declared type of X.

The first operation results in the evaluation of the dynamic address and its stacking along with information as to type.

'Take Expression' is an abbreviation for the object program which results in the value of the expression and its type being left in the stack.

'Store' then examines the stack, does any necessary type transformations, and stores the result of the expression at the given address.

2.4.2 Formal Assignments

When X is a formal parameter called by value it will be represented by information in the stacked link data (called a PARD), for the appropriate procedure activation. The mechanism for dealing with parameters to procedures is discussed in detail later, but in the present case it can be taken that the PARD gives the evaluated dynamic address of the actual parameter corresponding to the formal parameter called by value.

The above example becomes in the object program

```
Take Formal Address )          ( Take Formal Address
Take Expression     )    or    ( Take Expression
Real Formal Store   )          ( Integer Formal Store
```

Here type information is not given with the operation for evaluating the address of X. This is because a procedure body is translated without reference to procedure calls. Take Formal Address just results in finding, from the PARD, the address and type, of X, and stacking this information.

The Store operation carries the type information which is given in the specification of the formal parameter to which X, the actual parameter corresponds.

Take Expression once again, results in the stacking of the value and type of the expression.

The Formal Store operation, performs any necessary type transformations, firstly with regard to the type given with the operation, and then with regard to the type given with the address of X.

Thus a real actual parameter can correspond to a formal parameter specified to be of type integer, to which is assigned an expression of type real. The result of this is that the actual parameter receives the real rounded-off result of the expression. However, the case of converting integer to real and back again should be suppressed as it has no effect, and might even lose accuracy.

Non-formal assignments carry type information with the Take Address operation because, unlike formal assignments, it might be necessary to compile this operation before the declaration is met. This is dealt with by the method of chaining (described later), and it is preferable not to have to deal with the store operation as well as the Take Address operation. Thus a type independent store operation is used.

2.4.3 Assignment to Procedure Identifier

This uses one of two operations, according to the type given at the start of the procedure declaration.

Store Real Procedure Value, Store Integer Procedure Value

This places the result of the expression, after any necessary type change, in the position reserved in the link data of the current activation of the relevant procedure.

2.4.4 Note that in general, as is shown above, the operations are made as complete as possible to minimise the size of the object program. This of course means that the sub-routine complex will have a lot of entries.

2.5 Boolean Expressions

Logical values are represented by one word of zeroes, or ones. Boolean manipulation is carried out, in much the same way as arithmetic, by working on such variables, using the stack.

Relations are binary operations, whose result is a truth value in the top accumulator.

The relations = and ≠ must be absolute, even on real numbers. (i.e. no attempt is made to allow for round-off error).

2.5.1 If Clauses

If clauses use two implicit jumps

```
     U.J.     (Unconditional Jump)
and  I.F.J.   (If  False Jump)
```

I.F.J. inspects the truth value in the top accumulator, and jumps if it is false. The stack pointer must be decreased by one.

Thus conditional expressions and conditional statements are dealt with similarly.

```
if B then E1 else E2  becomes      TAKE B
                                   I.F.J.  b
                                   TAKE E1
                                   U.J.     a
                                b: TAKE E2
                                a:

and if B then S1 else S2  becomes      TAKE B
                                       I.F.J.   b
                                       S1
                                       U.J.     a
                                    b: S2
                                    a:
```

where a and b are program addresses.

An if statement is even simpler

```
if B then S   becomes    TAKE B
                         I.F.J.    b
                         S
                      b:
```

Implicit jumps never involve any change of level · they could never jump out of a block for instance. Hence the mechanism for a go to statement (dealt with later), which allows for such possibilities, is not needed, and implicit jumps just change the program counter.

2.6 Procedures and Blocks

The system to be used in KDF9 Algol is developed from that given in the article 'An Algol 60 Translator for the XI' by Dr. Dijkstra. [a]

Briefly, every procedure and block is given a Block Number, dependent on its lexicographical level (BN = 0 for the main program). At any given moment we wish to know what portion of the stack concerns the last activation of each block. (It is possible, because of the recursive nature of Algol for a block to call itself, either directly or indirectly).

a) Published in German in two parts (MTW, 2, 1961, pp 54-56, and 3, 1961, pp 115-119).

This information for the current block is called the parameter pointer (P.P.). When a block is entered, its Block Number (given in the object program) is set up in the stacked link data. The set of P.P's corresponding to the last activations of all current blocks are used to address all currently available quantities.

These P.P's are in the stacked link data, but are, for reasons of efficiency, also given in a table, called DISPLAY. DISPLAY gives a value of P.P. for values of Block Number (B.N.) up to the current B.N. The stacked link data also connects a block with its containing block by a dynamic chain, which takes in all activations of all current blocks, used for transferring control between blocks. The static, or lexicographic chain, also given in DISPLAY, is used for getting information from other blocks without transfer of control.

On entry to a block a set of places is set aside in the stack, for housekeeping, and for declared variables.

The amount of this storage is known, partly at compile time, and partly through evaluation of array declarations, and thus controls where the beginning of the working storage is. This information, also kept in the link data, is given by W.P. During the execution of statements the stack is used for anonymous intermediate results, under control of an accumulator pointer, A.P. Between statements A.P. has the value of W.P.

During the running of a program P.P. and A.P. are always available - other information can be obtained from the stack.

Take the example of a block <u>a</u> in which a procedure is called. The procedure body is block b, whose declaration is contained in the block c.

After completion of entry to block b, the link data is set up:-

APa	:	Procedure Result
PPb	:	PPc (Static Chain)
		PPa (Dynamic Chain)
		BNb (Block Number)
		WPb (Working space pointer)
		Return address to block a
		PARDS
APb, WPb	:	

The P.P. of the last incarnation of the lexicographically enclosing block (P.Pc) is used for the static chaining (also given in DISPLAY). Thus to find a declared variable, DISPLAY is used, rather than the static chain, to convert a Block Number, n, and a position p relative to the start of the block (P.P.) in which it is declared.

The dynamic chaining is given by the P.P. of the block from which the current block is entered (P.Pa)

These two chains must be set up at each call of a block or procedure. The current value of A.P. of the calling block is used to find the value of P.P. of the activation of block being called. Under control of this new value of P.P. the two links are set up,

a) The Dynamic Link is just the previous P.P.

b) The static link is set up, immediately after entry to the block, by the operation S.C. (Short Circuit) using DISPLAY

P.P. is used to find B.N, which has been set up by the calling mechanism. Then the entry in DISPLAY corresponding to one less than this value of B.N. gives the P.P. of the lexicographically containing block. This is the required static link.

2.6.1 Exit from procedures and blocks

The normal exit from procedures is by means of the operation RETURN. This operation uses the link data to update DISPLAY and to return control to the operation following the one that called the procedure.

The link data gives the P.P. of the block or procedure to which return is being made. Then display is updated by working back through the static chain, from this position given by P.P, and resetting DISPLAY, until the point where the static chain and DISPLAY are the same.

In the case of Blocks, or procedures called at their own level, or the pseudo-blocks made from 'for' statements, it is unnecessary to update display. This can be avoided at run-time by checking whether the last step in the lexicographical chain is the same as the last step in the dynamic chain.

Thus if the P.P. used by RETURN is the same as the entry in DISPLAY corresponding to B.N. - 1 update display can be avoided.

2.6.1.1 Jumping out of Procedures and Blocks

Evaluation of a label will result in obtaining a block number, and a value for the program counter - which keeps track of progress through the object program. The block number enables a value of P.P. to be obtained from DISPLAY. This P.P. is used to get a W.P. from the stack, and A.P. is set to this value of W.P. Finally control is transferred to the new value of program counter.

2.6.2 Calling Mechanism for Blocks

As stated earlier, blocks and procedures are treated alike - in fact a block is a procedure called only once, in the same position as its declaration.

Entry to a block is by the normal 'Call Procedure' operation, having specified the number of parameters to be zero. Procedures act as sub-routines and return to the operation immediately following 'Call Procedure'. This operation to which return is made is itself an implicit jump, which jumps around the procedure body (i.e. the block itself). This is necessary because, as stated earlier, in the case of a block, the declaration and call are at the same point.

The jump, around the block, reaches the operation REJECT, which deletes the accumulator which would have been used in the case of a type procedure called by a function designator.

A procedure body is automatically treated as a block. Only in the case when the procedure body is not actually a block need this special provision be made. There is no point in introducing an extra block level unless necessary.

2.6.3 Display

DISPLAY, the set of stores which duplicate the static chaining, uses a fixed amount of storage. The XI translator uses 32 stores, and hence 5 bits for 'n' in dynamic addresses. This has been found to be sufficient, but it might be worth increasing its size to 64 for KDF9. It is difficult to visualise a program falling foul of this limit, for entries are made to DISPLAY only for each lexicographical level of block or procedure.

2.6.4 Procedures and Parameters

In Section 2.6 it was stated that space was left, in the link data generated at a procedure call, for a set of PARDS, or static characterisations of parameters.

The PARDS are set up when a procedure is called, by transforming a set of PORDS - which correspond to the set of actual parameters. In particular PARDS corresponding to parameters called by value will contain the evaluations of the corresponding actual parameters. The object program generated from the procedure body will treat such parameters as local variables, whose position relative to the start of the link data of the current activation of the procedure is known. This object program will act as a sub-routine, called from various other places by procedure calls. Naturally on entry to a block, procedures declared at its head will be avoided by means of an implicit jump order.

2.6.4.1 Procedure Calls

A Procedure Call consists of a set of PORDS, corresponding to the actual parameters, and an operation 'Call Procedure'. PORDS can just be simple dynamic addresses, or could for instance point to implicit sub-routines for evaluating expressions.

The PORDS will be used by the call procedure operation, and will be avoided, by means of an implicit jump around them, in the running object program.

The operation 'Call Procedure' has already been described apart from its work on PORDS.

Call Procedure transforms PORDS into PARDS, which each use two words of storage. The set of PARDS immediately follow the link data. PARDS, like anonymous intermediate results in the stack, contain a word of identifying information. Thus PARDS can be used regardless of the fact that various kinds of PORD (i.e. actual parameters) can be used to set them up.

We now deal with the PORDS and PARDS for various kinds of parameters.

2.6.4.2 Simple Variable as Actual Parameter

In this case the PORD is complete in itself, and just gives the dynamic address (n,p) of the simple variable. As part of the calling mechanism the PORD-PARD transformation produces from this dynamic address in the object program, a static address (using DISPLAY) which is stacked as a PARD. This address evaluation must be done at procedure call because the dynamic address is given in terms of DISPLAY, as set up at procedure call. DISPLAY could vary within the procedure and hence invalidate the (n,p) of the PORD. The alternative, obviously inefficient, would be to maintain the dynamic address, and to reset DISPLAY at each use of the parameter. The PARD is also set up with identifying information.

2.6.4.3 Expression as Actual Parameter

If an actual parameter is an expression then an implicit sub-routine of object program operations is generated. A PORD is made which contains the starting address of the sub-routine, and an indication that the PORD refers to an implicit S.R.

Thus in the case of more than one such parameter the object program consists of

Implicit Jump to the Procedure Call, around the S.R's and PORDS
A set of implicit S.R's
A set of PORDS
The operation 'Call Procedure'

Due to the system of translation the PORDS are given in reverse order to the actual parameters.

Thus 'Call Procedure' works backwards through the PORDS, which are either complete in themselves, or point to implicit S.R's.

The PARD corresponding to a PORD pointing to an implicit S.R. consists of the address of the S.R., the P.P. ruling at the time of the generation of the implicit S.R., and, as always, identifying information. Thus when the parameter is called, the identifying information shows that an implicit sub-routine is to be used, and then the P.P. given in the PARD is used to set up the DISPLAY so that the sub-routine can be evaluated.

An implicit S.R. is a simple form of block. It is entered using the normal procedure mechanism, and left by the RETURN operation.

The operation E.I.S. (end of implicit S.R) uses a simplified form of Store Procedure Value. Store Procedure Value must store the result in the place given by the P.P. of the procedure which is obtained from DISPLAY. E.I.S. however, can just store the result under control of the ruling P.P. After doing this it then performs the function of RETURN.

Because of this simple form E.I.S, unlike S.P.V., does not need a block number.

2.6.4.4 Subscripted Variable as Actual Parameter

A subscripted variable as actual parameter is translated into an implicit S.R, whose last operation before E.I.S is INDA, which therefore delivers the address of the variable. This is necessary as the parameter may be called on either side of an assignment statement.

Thus it is possible to call on a formal parameter, by means of its PARD, and receive instead of the expected result, the address of the subscripted variable. This possibility is dealt with in Section 2.6.4.9.

2.6.4.5 Constant as Actual Parameter

This is dealt with in much the same way as an expression. For the sake of efficiency the actual constant is stored in the object program, as if it were an implicit sub-routine. The corresponding PORD gives the address of the constant, information specifying that the parameter is in fact a constant, and its type.

The PARD which is made from this PORD contains the actual constant, and a word noting this fact, and giving type.

2.6.4.6 Array as Actual Parameter

An actual parameter which is an array is described in its PORD with the (n,b) relating to the storage function. The PORD-PARD transformation just forms the physical address of this storage function.

2.6.4.6.1 Arrays Called by Value

After the PORD-PARD transformation, the PARDS corresponding to scalar parameters called by value are replaced by the value of the corresponding actual parameters. This is done by a set of 'Take Formal Result' operations.

The case of arrays called by value is somewhat more complicated. The system for this imposes some upper limit on the number of dimensions of the array. This is because, in the link data of the procedure which has an array called by value, we must erect space for the storage function of the array, without knowing its dimensions.

The KI translator has the arbitrary limit of 5 for the number of dimensions allowed in arrays called by value - the decision depends on how much space is throught reasonable to allow for the possibility of many - dimensional arrays.

Before processing the value list, the Call Procedure operation in performing the normal block entry will increase W.P to allow for working storage and set A.P.=W.P. This increase will have allowed a set amount for the storage functions of arrays called by value. Processing the value list, containing arrays, will now set up second order working storage in a similar way to obeying an array declaration.

The operation which assigns the value array has two parameters - leading to the PARD, and to the storage space set aside for the storage function. The PARD will lead to the storage function of the array which is the actual parameter. This storage function is used to make a storage function, using W.P. to give the start of the array, which is placed in the space set aside for it. Then the whole array is transported, element for element, if necessary converting type, under control of the original storage function, which gives type of the actual array, and the PARD, which, from the specification, gives type of the formal array.

After this is complete W.P. is moved on, by the physical extent of the array.

In this case we have an operation followed by two addresses, which does not fit into the usual pattern of operations and addressed operations. This does not matter however, since chaining will never be necessary at translation time. This is because both the position of the PARD, and of the space for the formal storage function will be known when the operation is generated.

2.6.4.7. Procedure Identifier as Actual Parameter

An actual parameter which is a procedure identifier could correspond to a type specification, or to a procedure specification. We wish to translate procedure calls independently of the procedure declaration.

Inside the procedure body, the PARD which will correspond to this procedure identifier could be called by

Take Formal Result (where specification has been say <u>real</u>)
or
Call Formal Function (where specification has been say <u>real procedure</u>)

Therefore Take Formal Result is only a special entry to Call Formal Function, equivalent exactly to C.F.F. for the case of no parameters.

The PORD corresponding to the procedure identifier will contain the start address of the procedure, and identifying information. The PORD-PARD transformation hands on this start address, together with the P.P. ruling at call time.

When a formal procedure is called (using operations Call Formal Function, or Call Formal Procedure), any PORD-PARD transformations necessary are made before updating display with the P.P. in the PARD of the formal procedure. Then all that remains is to jump to the start address given in the PARD.

2.6.4.8 Formal Parameter as Actual Parameter

This is best illustrated by an example

```
procedure  P(u);

              begin
                      ; Q (u) ;
              end
```

Inside P,u is given by the block no. of P and the position of u . This allows reference to the PARD of u, which is in the link data of P. In connection with the call of Q we make a PORD for u, which in this case gives the fact of formality, and the position of the PARD of u in the link data of P.

The PORD-PARD transformation made in calling Q just copies the original PARD into the new PARD in the link data for the call of Q.

Thus the PORD-PARD transformation always evaluates the dynamic address, and then checks for formality.

2.6.4.9 Formal Recursion

A PARD can consist of

a) Address of a Program, and its type

b) A constant, and its type

c) Address of a program to jump to.

Type c is the one that gives rise to formal recursion (by a procedure, or an implicit sub-routine). Formal recursion involves setting up another block in the stack. To avoid stacking the fact of formality in the link data it is necessary to make sure that the exit of a formal recursion needs no extra information. This exit, will be by means of the operations RETURN, which is used in implicit sub-routines as well as procedures.

The fact that all accumulators carry an indication of their meaning lets us allow for the fact that a formal recursion might result in an address, when a result was expected.

This can happen in the case of a subscripted variable as actual parameter, for the implicit sub-routine must use INDA (Index Address) rather than INDR (Index Result). This is because the PARD could be used for either side of an assignment statement.

We introduce the operation

C.T.R. (Conditional Take Result)

C.T.R. inspects the top accumulator, and if it is an address replaces this by the value contained in this stack address.

The operations which involve Formal Recursion when PARD is of type c are,

Call Formal Procedure	(C.F.P.)
Call Formal Function	(C.F.F.)
Take Formal Result	(T.F.R.)
Take Formal Address	(T.F.A.)

All of these use the same routine for entering a formal recursion, and must use the same exit. This routine could, in the case of an implicit sub-routine, perform short circuit, rather than include this in every implicit sub-routine.

Thus Call Formal Procedure is just the operation Call Formal Function followed by REJECT.

Take Formal Result and Take Formal Address make special entries to C.F.F. which sets the number of parameters to be zero.

Take Formal Address is in fact the routine for entering a formal recursion.

Take Formal Result is T.F.A. followed by Conditional Take Result.

Call Formal Function is bound to result in a value in the top accumulator, so C.T.R. is unnecessary for it.

2.6.4.9.1 Formal Call of a Procedure by a Statement

Take an example

```
procedure Q (u); real  u :

                 begin
                       ; u ;
                 end
```

called by Q (P(a,b)) where P is declared as a function designator.

The call of u in the body of Q is by the operations

Take Formal Result
REJECT

2.6.5 Function Designators

Section 5.4.4 of the Algol Report is interpreted as follows.

For a procedure to define a function designator it must be given a type and within must appear, explicitly, at least one assignment to the procedure identifer. At run-time at least one such assignment must be obeyed.

Such an assignment can occur in a lower, block or procedure. Hence the operation 'Store Procedure Value' (Section 2.4.3) is accompanied by a block no. This is used to get P.P. from DISPLAY, which points to the store address used for the procedure result.

Any block or procedure sets up the accumulator for a result, only procedures used as function designators do not use the operation 'REJECT' to delete this space.

2.7 Labels

A label occurs in a given block, and always points to a fixed point in the object program pertaining to this block - but this label has as many values as a block has activations.

Addressed operations can include either (n, p), or a program address. We do not wish to have operations which carry both Block Number and program address - for reasons of storage efficiency. Thus we use a system of stacking program address and Block Number. This can then be found by the normal dynamic address.

Then go to L generates

```
Take Label
Jump to Accumulator
```

Take Label carries a dynamic address which is used to get the program address and Block Number from the stack to the top accumulator.

Jump to Accumulator checks whether the block number part of the accumulator contents has a DISPLAY entry which is equal to the present ruling value of P.P.

If not it performs update DISPLAY and 'Go To Adjustment'. G.T.A. uses the new P.P. to reset A.P. from the link data.

This system of having labels in an accumulator immediately makes Designational Expressions similar to Arithmetic or Boolean Expressions.

The method of setting up labels in the stack is as follows.

At the end of the object program corresponding to the block proper, after the operation RETURN, is a set of operations which take program addresses, add current Block No, and assign the result to the positions reserved in first order storage for the labels.

Thus after entry to a block, an implicit jump is made, past the RETURN operation, to the label - setting instructions. These finish with an implicit jump back to the start of the block proper.

Then it is these positions in first order storage whose dynamic address is given on the Take Label operations.

2.8 Switches

A Switch is treated as a procedure with one parameter, which serves to pick a Designational Expression from a linear array.

i.e. Switch S := A, B

is thought of as

```
label procedure  S (n);
        S:= if n = 1 then A else if n = 2 then B
```

The evaluation of a switch can 'go recursive' in two ways

a) A Designational Expression can include a Switch Designator

b) In a Designational Expression can be a function designator which contains go to Switch

Thus evaluation must be capable of complete recursion. A set of recursions of the first type would result eventually in finding a label, and then a set of RETURNS would have to be worked through. It does not seem worthwhile trying to combine these into a 'Master Return' since the possibility of the second type of recursion, which has to go back to the function designator, has to be allowed for.

2.9 Representation of the Object Program

The representation of the object program in core storage is such as to allow unchaining, needed for parallel declarations and non-local variables.

The Object Program consists of

a) Operation

b) Operation + Block No. + Stack Address (relative to P.P.)

c) Operation + Program Address

d) PORDS

e) Constants

f) Operation + 2 dynamic addresses

The elements which can require chaining are types b, c and d.

PORDS will have to be treated specially as they might need block number, program address, and identifying bits. However types b and c can use the same type of storage.

Type f (used for setting up value arrays) is not involved in chaining, so is not so important.

We have avoided having operations with block number and program address because the program address will be in terms of KDF9 syllables rather than words. This allows the object program to be packed by syllables.

The block number n can take at most six bits. Thus two 8 bit syllables will be sufficient for a dynamic address (n, p) or a program address. This will be the space used for chaining links. The operation itself can fit into one syllable.

PORDS will need 4 syllables. The identifying information can be used during chaining to differentiate PORDS from addressed operations by using a corresponding bit in operations.

The PARDS produced at procedure call and stacked in link data consist of the static address, the P.P. ruling at call time, and identifying bits.

2.10 Run-time Error Print Out

Inconsistencies at run-time are likely to arise from two sources. The first due to irregularities in the Algol, which would not be checked at compile time and the second due to failures in the arithmetic.

The first type include the following:-

a) The lower bound is greater than the upper bound in an array declaration.

b) The number of subscripts not compatible with the number of dimensions given in the corresponding array declaration.

   Here information in the storage mapping function is checked against the number of values at the top of the stack when INDA or INDR is met.

c) The evaluated mapping function does not give information about an element within the array.

d) The actual parameters are not compatible with the formal parameters.

   The number of actual parameters is given by the difference between P.P & [illegible]. which covers link data and two words per PARD.

Some of the above irregularities could be found at compile time (i.e. types b and d). However with the one pass compilation along with the method of chaining described in this report it would be difficult to do so then.

The second type arise from numbers becoming overlength and also due to values, which have been changed from integer to semi-real, being assigned to an integer, used in a relation or in a subscript evaluation.

When an inconsistency has been found a message will be printed out. The difficulty is to pin-point the error in terms of the Algol and not in terms of the object program which would be meaningless to the programmer. To do this two tables are constructed at compile time and stored, probably with the two tables intermingled, on magnetic tape. The two tables are:-

a) A table of the program count against the Algol line number.

   Details can also be given of the values of the program count where failure is likely to occur along with the type of failure to be expected.

b) A table of explicit block counter against the name of the block or, if anonymous, the type of the block. Such block counters, found lexicographically, enable each block to have a unique number which is stacked at each incarnation of the block. This table could be expanded to include more than one block (including implicit sub-routines for calculating parameters, for statements and procedures) on a line and giving details as to their number, type and identifier, if appropriate.

If a failure occurs at run-time the program counter is used to obtain information from the first table whilst the block counter, which is contained inthe link data of the current block and is found using P.P, is used to obtain information from the second table.

3. The Compiler

The compiler will use a one pass system, employing a name list and a translation stack. The name list will contain the names and details of, all identifiers with a currently valid declaration.

A system of chaining, through the object program being generated in the core storage, will be used to avoid extra passes. The translator stack will be used to re-order operations according to a priority table.

Only a minimum amount of Algol will be held in storage at any one time and to give information about the Algol which has passed a system of state variables and stacked obligations is used. Thus it is possible to discriminate on each individual delimiter instead of the combination of two consecutive delimiters.

Checking routines are to be used to ensure legality of the Algol input.

3.1 Input Routine

The Algol input to the compiler is regarded as the printed page rather than the tape. For this reason carriage return, without an accompanying line feed, and back space are forbidden. In order to allow for the possibility of joined tapes a case definition character must come before the first case dependent character after blank tape.

There is a unique hardware representation for each Algol symbol.

The routine looks for and counts string quotes and packs the characters between them, without suppressing any symbols such as CRLF or space, and stores them, for use by a procedure body, as an implicit sub-routine.

Outside string quotes such characters as CRLF and space can be ignored as can the various comment conventions, after checking that they have not been used illegably such as <u>comment</u> following any character other than <u>begin</u> or ; except when it is inside a comment.

The routine will look for and count <u>begin</u>'s and <u>end</u>'s to look for the end of the program.

CRLF characters are counted for use by the error print out routine.

The basic cycle routine will read until the next delimiter and then find if a name or a constant has passed so that the routine could act accordingly.

The delimiter is then examined (possibly using knowledge as to the Algol that has passed, to differentiate between two different uses of a left round bracket say) and the routine jumps to the appropriate sub-routine.

After meeting <u>procedure</u>, for instance, a sub-routine would deal with the whole procedure heading following, whilst using the basic cycle routine to deliver the Algol input.

In this way the legality of the Algol can be checked.

3.2 Name List

A list of the names used for identifiers is built up as compilation proceeds. Along with each name is stored a word of information giving details of the identifier. This name list is also used by the checking routines. By making the internal representation of letters and digits lie in the range 0-61 the names can be packed, as 6 bit characters, eight to a KDF9 word. Thus a name will be limited to a maximum of 8 characters. If, however, a name contains more than 8 characters only the first 8 will be used as the name and a message will be printed out to warn the programmer of this.

The names should be packed into the KDF9 word in such a way that one can differentiate between X10 and X1 which differ only by the addition of zero on to the end of the first. The method of packing these names is shown below.

0	1	X	0	0	0

1	X	0	0	0	0

3.3 <u>State Variables</u>

State variables are used to give information about what has passed, which is necessary as only a small amount of the Algol program will be in the machine at any one time.

e.g. The state variables E.F. has a value 1 if in an expression and 0 if at statement level.

The state variables also explain the meaning of a character using knowledge of what has past, e.g. state variable for a , not in statements has the following values.

1) no, expected and therefore lead to fail

2) between for list elements

3) after (

4) after [

also the state variable V could have the following values:-

V = 0	at <u>begin</u>
V = 1	when a statement is found
V = 2	when a declaration is found

Thus V can indicate a compound statement (0→ 1) a block (0 →2) or lead to fail routine if (1→ 2). This variable can be used to indicate the first statement of a block; this information is required to indicate the end of the declarations.

The state variables will be stored in fixed locations but current values of the variables may be stacked, probably in packed form, from time to time when necessary and then unpacked when unstacked.

3.4 <u>Compile Time Error Print Out</u>

It is proposed that compilation should cease when the first inconsistency is found and a message be printed out. It is doubtful whether it would be economic to proceed with the compilation having made assumptions as to the error and as to what should be changed to correct the program.

To pin-point the inconsistency it is suggested that the message printed out should contain the following:-

a) Line number

b) Error type (i.e. no comma allowed here)

c) The next 100 characters or 2 lines of Algol program (counting only non-trivial information).

   Having kept an account of the position on the page this can be printed out to reproduce exactly the Algol input.

d) The last label and the last procedure identifier declared.

e) The last few items of the translator stack.

3.5 Method of Chaining

The method of chaining is used to allow a one pass compiler to deal with forward jumps and the possibility of an identifier being used before its declaration information is available.

Thus chaining will be used to complete addressed operations and PORDS in the object program.

3.5.1 Use of Identifier (See 3.5.5)

When an identifier is met, the name list of the current block is searched for its declaration. If this is found the information about the declaration is obtained. However, if there is no declaration in the name list then:-

a) There is no entry in the current name list for this identifier. This means that it can be non-local or local but not yet declared. A "non-allocated" entry is made in the name list noting the position (α) reached in the object program which is left blank.

b) There is already a "non-allocated" entry in the name list. Take the object program count (α) from this entry and put it into the current space (β) of the object program and put β into the "non-allocated" entry.

3.5.2 Declaration of Identifier (See 3.5.5)

At declaration the current name list is searched with the following results:

a) If there is no entry for this identifier in the list a normal entry is added.

b) If there is already a normal entry for this identifier in the current list lead to fail routine.

c) If there is a "non-allocated" entry in the list for this identifier it is normalised and the object program must be brought up to date concerning the declaration information.

   To do this the "non-allocated" entry contains the last place in the object program requiring this information. Similarly this place in the program contains the address of the next place requiring the information and so on until the end of the chain which does not lead to any further places in the program.

3.5.3 End of Block (See 3.5.5)

When leaving a block there may be some "non-allocated" entries left and the name list for the surrounding block is searched for entries for these identifiers with the following results:-

a) If there is no entry in the surrounding block list copy the "non-allocated" entry into it.

b) If there is already a normal entry follow the chain procedure described in (c) above to put the declaration information in the program generated.

c) There is a "non-allocated" entry in the list of the surrounding block pointing to position γ' in the program. Taking the program address given in the name list entry of the block to be left chain through these program spaces and add γ' to the end of this chain. Thus one chain is made from the two separate ones. i.e.

```
a :=      +a      -a      +a      a :=      a :=          real a
α'        β'      γ'      α       β         γ
```

The "non-allocated" entry now replaces that already in the name list of the surrounding block.

The name list of the block to be left is now collapsed.

3.5.4 End of Program

At the end of the program the only "non-allocated" entries left belong to identifiers used without declarations and can indicate library procedures or illegal Algol input.

3.5.5 Note

Subscript bound expressions can only depend upon variables or procedures which are non-local to the block for which the array declaration is valid.

In this case the chaining method described above must be modified as follows:-

3.5.5.1 Use of Identifier

If, when translating a subscript expression, an identifier x is found then:-

a) If there is a normal entry for x in the current list lead to fail routine.

b) If there is a "non-allocated" entry for x in the current list add a subscript bound marker to it, if not there already.

c) If there is no entry for x in the current list add a "non-allocated" entry with the subscript marker set.

3.5.5.2 Declaration of Identifier

When an identifier is delcared the current name list is searched as in the chaining method but condition c of 3.5.2 has to be modified as follows.

3. If there is a "non-allocated" entry already in the list lead to fail routine if it has the subscript marker set otherwise normalise the entry and use the chaining method already described in 3.5.2 to place the declaration information in the program.

3.5.5.3 End of Block

At the end of the block, any "non-allocated" entries which are added to the list of the surrounding block have the subscript bound markers deleted.

3.6 Translation Stack Priorities

In order to deal with the translation of the Algol whilst only having a small part of it available at any one time a translator stack is used. This sorts out questions of arithmetic precedence, implicit jumps etc. and is used as a temporary store for operations, obligations and also for the current values of state variables. These are stacked along with a stack priority (spr). When an operation or delimiter is met from the input its compare priority (cpr) is used to decide whether some items from the stack should be dealt with first.

The stack is emptied whilst the top item has a stack priority (spr) > compare priority (cpr) of current operation.

In order to determine the values of the stack and compare priorities the following points were among those considered.

a) spr of <u>begin</u> is the lowest possible, and can thus be zero

b) cpr of everything must be higher than zero so that <u>begin</u> can never be unstacked automatically (i.e. by means of priority)

c) cpr of <u>begin</u> <u>if</u> (or [are void since they go straight into the stack without first emptying any items already stacked

d) <u>then</u> E controlling an expression or <u>then</u> S controlling a statement must unstack back to the corresponding <u>if</u> which they replace.

e) <u>else</u> E or <u>else</u> S controlling an expression or a statement respectively unstack back to the corresponding <u>then</u> which they replace.

f) Between <u>if</u> and <u>then</u> will be a Boolean which might be conditional. To clear this the spr of <u>else</u> E must not be less than the cpr of <u>then</u> E or <u>then</u> S.

g) the cpr's of ; or <u>end</u> must be equal to 1 to unstack everything back to a <u>begin</u>

h) A set of <u>else</u> S are terminated by ; or <u>end</u> which must unstack through them.

i) <u>else</u> S must unstack through an unconditional statement which might involve a conditional expression, whose <u>else</u> it must clear.

j) To avoid complications in the case of a for statement appearing inside a conditional statement i.e.

<u>if</u> B_1 <u>then</u> for $V = A$ <u>do</u> <u>if</u> B_2 <u>then</u> S_1 <u>else</u> S_2 <u>else</u> S_3

it is suggested that <u>begin</u> and <u>end</u> are introduced around the for statement (which is treated as a block).

Thus the following equations and inequalities are deduced.

spr (begin) = 0

crp (begin) = cpr (if) = cpr (( ) = cpr ([ ) = void

spr (if) = 0

spr (if) < cpr (then E)
spr (if) < cpr (then S)

spr (then E) < cpr (else E)
spr (then S) < cpr (else S)

spr (else E) ⩾ cpr (then E)
spr (else E) ⩾ cpr (then S)

cpr (;) = cpr (end) = 1

spr (else S) ⩾ cpr (end)

spr (else E) ⩾ cpr (else S)

spr (then S) ⩾ cpr (;)

spr (else S) < cpr (else S)

spr (do) ⩾ cpr (;)
spr (do) ⩾ cpr (else S)

spr (( ) = spr ([ ) = 0

spr ( )) = spr ( ]) = void

These, and similar formulae, give rise to the following table of priorities.

	spr	cpr
begin	0	void
end ;	void	1
if	0	void
then S	1	3
then E	0	3
)	void	3
]	void	3
(	0	void
[	0	void
else S	1	2
else E	3	2
do	2	void
:=	3	13
≡	4	4
⊃	5	5
∨	6	6
∧	7	7
¬	8	8
> ⩾ ⩽ < ≠ =	9	9

	spr	cpr
+ -	10	10
x / ÷ NEG	11	11
↑	12	12
IND	13	void

3.7 Statement Translation

Algol statements are translated by the compiler into the object program which is interpreted by the sub-routine complex at run-time. The object program contains operations, with or without address, PORDS and constants.

Since only a small part of the Algol will be in the machine at any one time it will be impossible to generate a complete object program as the Algol program is traversed. This is because sufficient knowledge to complete an operation may not be to hand until more of the Algol has been dealt with. Use is thus made of the translation stack so that obligations can be completed later in the translation.

3.7.1 Conditional Statements

The Boolean expression in the if clause allows a choice to be made at run-time as to whether the statement, or which one of two statements, following is to be obeyed.

i.e. if B then S_1 ; or if B then S_1 else S_2 ;

Two implicit jumps are generated by the compiler to jump to and round the statements concerned.

They are:-

a) U.J. which is an unconditional jump.

b) I.F.J. which jumps if the top accumulator contains the value false

In the case of if B then S_1 else S_2 the object program resulting is

```
    TAKE B
    I.F.J. - β
    S1
    I.J. - γ
    β: S2
γ:
```

On meeting then it is known that an implicit jump will be required to jump around statement S_1 if B turns out to be false, but as the extent of S_1 is not known it cannot be completed at once. Thus a space is left in the object program for the jump and the obligation to complete it as soon as possible is stacked. When else is met this jump can be completed and the space in the object program filled in.

Similarly the implicit jump U.J, to enable the statement S_2 to be bypassed after obeying S_1, is stacked and then completed at the end of the conditional statement.

3.7.2 For Statements

For statements are made into blocks even though the controlled statement may itself be already a block.

The Algol for clause is translated into a set of entries into the for complex which is part of the sub-routine complex used by the object program at run-time. Also included are some implicit unconditional jumps which again make use of the stack to store the obligation to complete them as soon as possible.

Communication between the object program and the for complex is achieved by means of two links called link L and link Σ

Link L is used to denote the current for list element.

Link Σ is used by the for complex as the return to the for complex after having used part of the object program as a sub-routine. This happens when calculating a for list element of the following type

V : = A <u>step</u> B <u>until</u> C

and the controlled variable V is required as both an address and as a value in more than one place i.e. V : = V + B.

An idea of the object program resulting from a for statement is given below.

<u>Algol</u>	<u>Object Program</u>	<u>Remarks</u>
<u>for</u>	go to Fa	Implicit unconditional jump to Fa to set up a block for the for statement.
V	Fb: TAKE ADDRESS of V	
	FOR 1	Obeys link L to take the current for list element.
	Fa: Instructions to enter block at Fd.	
for list elements		for list elements given as entries to the for complex (see 3.7.2.1)
	go to Fe For 8	Instruction placed at the end of the for list elements to jump past the for statement after completing the block exit.
<u>do</u>	Fd: For 0 go to Fb go to Fb	Completes setting up of the block and reserves space for link Σ Then jumps to Fb to work through the for statement.

Fc: Statement	
go to Fb	Implicit conditional jump back to the for clause.
Fe:	Completion of for statement.

The various kinds of for list elements are translated as follows:

3.7.2.1 Arithmetic Expression Element E_1 ,

Object Program	Remarks
TAKE E_1	Program to evaluate E_1
FOR 2	Stores value of E in the address of the controlled variable V and puts the position of the next for list element into link L. Then jumps to Fc.

3.7.2.2 While Element E_2 while B ,

Object Program	Remarks
TAKE E_2	Program to evaluate E_2
FOR 3	Store value of E_2 in the address of the controlled variable V.
TAKE B	Program to evaluate B
FOR 4	Look at the top accumulator, which contains B. If true jump to Fc keeping link L unchanged so that this for list element is still current.
	If false put the position of the next for list element into link L and jump to Fb.

3.7.2.3 Step-until Element E_3 step E_4 until E_5

Object Program	Remarks
TAKE E_3	The important thing here is that the increment shall not be added on to the controlled variable for the first time. Thus the first time V := A and thereafter V := V+B
FOR 5	
TAKE E_4	
FOR 6	The routine "TAKE ADDRESS OF V" is used to find the value of V needed to calculate its new value and for this reason link Σ is used by the for complex.
TAKE E_5	
FOR 7	

Object Program	Remarks
	If $(V \cdot E_5) \times \text{Sign } E_4 > 0$ jump to Fb after putting the position of the next for list element into the link L, if not jump to Fc keeping link L unchanged. Entries FOR 5, FOR 6, FOR 7 serve to show the extents of E_3, E_4 and E_5.

Links L and Σ, which give a complete picture of the state of completion of the for list elements, are kept in the stack so that the for complex can be used recursively (e.g. during the evaluation of an element a procedure call might involve another for statement).

3.7.3 Procedure Statements

For each actual parameter a PORD is generated which gives details about the parameter. The PORD may be complete in itself if the actual parameter is a simple variable or it may indicate the starting address of an implicit sub-routine, mapping function or program address if the parameter is an expression, array or a constant respectively.

After dealing with a procedure identifier the next delimiter will indicate whether a parameter list is to follow. If it is a left bracket the program count is noted after leaving a space in the program for an implicit jump around the implicit sub-routines and the PORDS. At the next delimiter the program count is again checked and if it has changed since the last delimiter an implicit sub-routine will have been generated and the PORD for that parameter can give the starting address of it. If this parameter delimiter does not indicate the end of the parameter list the process is repeated for the next parameter.

It should be noted that the parameter delimiter could be a comma or

) < letter string > : (

The PORDS, one of which is generated for each parameter, are stacked whereas the implicit sub-routines or constants go straight into the object program. At the end of the parameter list the PORDS are unstacked into the object program and they will thus be in reverse order. The implicit jump around the implicit sub-routines and the PORDS is completed and an operation CALL PROCEDURE, which contains details as to the number of parameters, is added to the program.

A picture of the object program generated by the following procedure call

P (E_1, 1.2, E_2, A, B) where E_1, E_2 are expressions and A, B simple variables is given below:-

Object Program	Remarks
go to Fa	Implicit jump around the program generated to evaluate the parameters and their PORDS.

Object Program	Remarks
α : TAKE E_1	Implicit sub-routine to evaluate E_1
β : 1.2	The value of the constant is placed in the program.
γ : TAKE E_2	Implicit sub-routine to evaluate E_2
PORD	Information about 5th parameter B
PORD	Information about 4th parameter A
PORD	Information about 3rd parameter and also pointing to γ
PORD	Information about 2nd parameter and also pointing to β
PORD	Information about 1st parameter and also pointing to α
Fa: CALL PROCEDURE	Operation to jump to the procedure body.

The implicit sub-routines used to evaluate expressions or subscripted variables as actual parameters are translated as blocks and are given a block number one higher than that ruling at the procedure statement.

3.8 Procedure Bodies in Code

The Algol is translated into an object program which is itself obeyed interpretively.

At no time does a KDF9 machine code version of the Algol exist and so code procedure bodies cannot be merely handed over intact to the object program for use at run-time.

They must be stored apart from the object program, probably on tape, either in binary form having been compiled before hand or in user code if it is possible for the sub-routine complex to call in the user code compiler as a sub-routine.

In the X1 translator procedures having their bodies in code are limited to library functions and are written by those people at Mathematisch Centrum having knowledge of the inner workings of the Algol translator.

3.9 Subscripted Variables

After generating program to evaluate a subscript expression the operation INDA or INDR must be inserted depending upon whether the address or the value of the subscripted variable is required at this stage. It is not always possible to know which is required when the closing bracket of the subscript expression is met and so the imcomplete operation IND is stacked under a high priority. The next delimiter recovers this from the stack and it can be completed and inserted in the program. To illustrate this two examples are given,

a) a [ ] +

At + IND is unstacked and as it is now known that INDR is required this can be stored in the program.

b) a [ ] : =

At : = IND is unstacked and the operation INDA is stored in the object program.

3.10 Procedures and Blocks

Blocks and procedures are treated similarly as a block is considered to be a procedure without parameters which is only called at one place.

At runtime the stack for a block, or a procedure, contains,

a) Link data, of constant size

b) First order information, for scalars, labels, mapping functions and also for PARDS in the case of a procedure.

c) Second order information, for arrays

c) Anonymous results

The extent of the link data and first order information is known at compile time when the end of the block is reached, and a state variable L is used to keep a count of the storage required by these. The final value of L, for each block, is used to denote the starting position for the storage of the array elements as the second order information.

If a block is left before it is completed, by meeting another <u>begin</u> or a procedure declaration, the current values of the state variables V (see 3.3) and L are preserved in the stack.

In order, at runtime, to have the values of the labels in the stack we introduce at compile time, for every block as many local variables as there are labels in the block.

The compiler generates a set of assignments to the positions first order information set aside for the labels and these are placed in the object program at the end of the block after the operation RETURN. Implicit jumps are also generated to lead to these assignments from the start of the block and then back to enter the block proper. Thus when the block proper is entered the first order stores for the labels will contain block number and program address.

3.10.1 Formal Parameters called by Value

The operation TAKE FORMAL RESULT is placed in the object program for each of the parameters called by value. This will cause at run-time the corresponding PARDS to be evaluated and replaced by their values before entry to the procedure body.

Since these parameters called by value are then to be treated as local to the procedure body their entries in the name list are changed from formal to local identifiers.

3.11 Speed of Translation

The probable relative speed of the KDF9 translator to the X1 translator will at first sight be low because of all the chaining involved in parallel declarations and non-local variables. However the method has a corresponding gain in that the name list is only ever searched down to the start of the current block. Another possible increase in speed is due to the fact that numbers are stored in the object program whenever they are read in regardless whether the number has been used before.

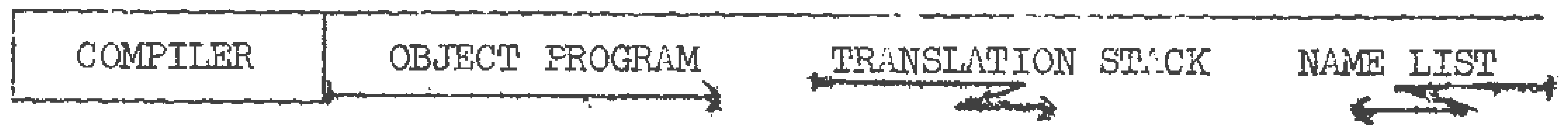
3.12 Use of Core Storage by the Translator

KDF9 translator will need core storage for:-

3.12.1 Compile Time

Compiler	fixed
Object program	growing
Translation stack	pulsating
Name List	pulsating

The following diagram shows how the available core storage is used at compile time.



The object program is placed next to the compiler, which is a fixed size.

The name list starts at the other end of the core store and pulsates towards the object program.

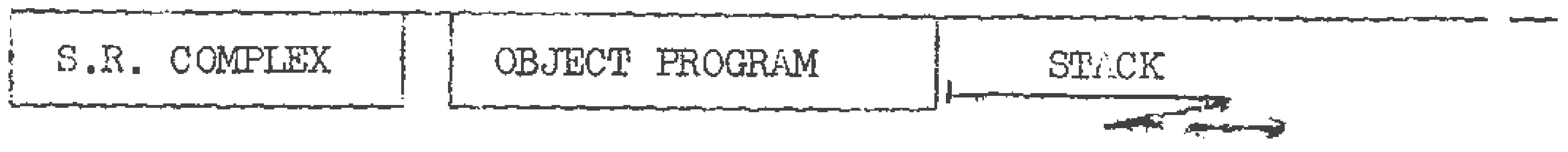
The translation stack, which in general will be smaller than the name list, is placed in the centre and pulsates towards the name list. If the object program reaches the bottom of the stack or if the name list and the stack meet the stack is moved, say 32 places, in the appropriate direction (and failing if this is not possible).

This means that positions in the stack are not fixed throughout the compilation and indeed the starting position of the name list can vary from one run to the next due to different sizes of machine or due to time sharing.

Great care must be taken to allow for this and especially to allow for the stack having moved between two references to it. The original starting position of the stack is chosen so that for reasonable sized programs it will not need to be moved.

3.12.2 Run-Time

Sub-routine Complex	fixed
Object program	fixed
Stack	pulsating



Since the compiler and the sub-routine complex are expected to be of roughly the same size the object program can be at the same position during runtime as it was when being compiled.

The run-time stack is then placed immediately after the object program.

4. Input and Output on XI

Some of the procedures used for XI Algol are described below.

FLOT (n, x)	This prints x as a floating number with n decimal places in the mantissa and a 2 digit exponent.
FIXT (n, m, x)	Prints x as a fixed point number with n digits before the point and m digits after it. If m = 0 the decimal point is suppressed.

The above two routines give a correctly rounded result. Underflow produces 0 and overflow produces + INF or - INF.

FIXP (n, m, x)	This routine will output x onto paper tape in a similar form to FIXT (n, m, x).
FLOP (n, m, x)	Outputs x onto paper tape on a floating number with n decimal places in the mantissa and m digits in the exponent (where m = 1, 2 or 3).

FIXP and FLOP all have automatic insertion of CRLF before any number for which there would be no room left on the present line of type. Thus they are ideal for intermediate output which may be used again as input and they can also be tabulated.

PRINT TEXT	Outputs typewriter symbols only as this routine uses the monitor typewriter
PUNCH TEXT	This routine will accept everything except basic symbols which are underlined words. The text, produced by PUNCH TEXT, enclosed by apostrophes act as a number separator when read in as data.

Other number delimiters are the sign of the next number (hence there is no output sign suppression), comma, C.R.L.F., two or more spaces after a digit or TAB. TAB is fixed, by convention, to be 8 places.

To avoid wastage of machine time on a program which uses up slowly a hand prepared tape it is recommended that a fast pre-run be used which could check decimal punching and form sum checks.

5. Acknowledgements

The authors are pleased to acknowledge the assistance received from Dr. E. W. Dijkstra, during their visit to Mathematisch Centrum, during December, 1961, particularly in that, as this report shows, Dr. Dijkstra was not content to merely describe the successful translator for the X1 computer, written by himself and Dr. Zonneveld, but spent considerable time considering the characteristics of the KDF9 computer, and the requirements for the A.P.D. translator.

Appendix — Abbreviations used in the Report

1. Operations in the Object Program

C.F.F.	Call Formal Function
C.F.P.	Call Formal Procedure
C.T.R.	Conditional Take Result
E.I.S.	End of Implicit Sub-routine
I.F.J.	If False Jump
IND	Index
INDA	Index Address
INDR	Index Result
J.T.A.	Jump to Accumulator
S.P.V.	Store Procedure Value
T.F.A.	Take Formal Address
T.F.R.	Take Formal Result
T.L.	Take Label
U.J.	Unconditional Jump

2. General

A.P.	Accumulator Pointer
B.N.	Block Number
P.P.	Parameter Pointer
W.P.	Working Space Pointer
PORD	Characterisation of Actual Parameter in Object Program
PARD	Characterisation of Actual Parameter in Stack
(n, p)	Dynamic Address
L.V.C.	Local Variable Counter
G.T.A.	Go To Adjustement
S.R.	Sub-routine